Inherently Flexible Software

Steven James Glover

Ph.D. Thesis

Research Institute in Software Evolution
Department of Computer Science
University of Durham

2000

ABSTRACT

Software evolution is an important and expensive consequence of software. As Lehman's First Law of Program Evolution states, software must be changed to satisfy new user requirements or become progressively less useful to the stakeholders of the software. Software evolution is difficult for a multitude of different reasons, most notably because of an inherent lack of evolveability of software, design decisions and existing requirements which are difficult to change and conflicts between new requirements and existing assumptions and requirements.

Software engineering has traditionally focussed on improvements in software development techniques, with little conscious regard for their effects on software evolution. The thesis emphasises design for change, a philosophy that stems from ideas in preventive maintenance and places the ease of software evolution more at the centre of the design of software systems than it is at present. The approach involves exploring issues of evolveability, such as adaptability, flexibility and extensibility with respect to existing software languages, models and architectures. A software model, SEvEn, is proposed which improves on the evolveability of these existing software models by improving on their adaptability, flexibility and extensibility, and provides a way to determine the ripple effects of changes by providing a reflective model of a software system.

The main conclusion is that, whilst software evolveability can be improved, complete adaptability, flexibility and extensibility of a software system is not possible. In addition, ripple effects can't be completely eradicated because assumptions will always persist in a software system and new requirements may conflict with existing requirements. However, the proposed reflective model of software (which consists of a set of software entities, or abstractions, with the characteristic of increased evolveability) provides trace-ability of ripple effects because it explicitly models the dependencies that exist between software entities, determines how software entities can change, ascertains the adaptability of software entities to changes in other software entities on which they depend and determines how changes to software entities affect those software entities that depend on them.

Contents

List of Figures

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Figure 8

List of Tables

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 7

Chapter 8

Declaration

No material contained in this thesis has previously been submitted for a degree in this or any other university.

Statement of Copyright

Acknowledgements

This thesis was sponsored by a CASE studentship from the Engineering and Physical Sciences Research Council (EPSRC) and British Telecommunications PLC.

First of all, I wish to thank Professor Keith Bennett for his invaluable advice and support during the course of my Ph.D. research. Thanks also go to my supervisor at British Telecommunications' Research Laboratories at Martlesham Heath, Steve Corley.

In addition, discussions with James Ingham, Paul Warren, Peter Young, Simon Smith, Stephan Jamieson, Claire Knight and Mark Jarvis of The Research Institute for Software Evolution have been an invaluable part of my Ph.D. study, providing both insights into my Ph.D. work and the wider areas of computer science and software engineering.

Lastly, thanks and love to my family for their support and encouragement throughout the course of the Ph.D.

To Jim, Val and Anelene.

Chapter 1

Introduction

There are two overall approaches to tackling the world-wide software maintenance and evolution problem. One way is to continue with existing approaches that concentrate mainly on the *process* of software maintenance, which involves system comprehension and reverse engineering. Another way is to concentrate more effort on finding radical new ways of writing software to deal with software maintenance and evolution problems, that is, focus on product rather than process issues. This is design for evolution and the emphasis for these types of approach is on *how* the software should be constructed in order to ease the software evolution task.

The process of software evolution typically consists of the following steps:

1. Identify the need for change;
2. Create a set of specifications that meet the new requirements;
3. Map the specifications to changes in the software, the ease of which is determined by the availability of constructs to model the requirements;
4. Make the changes to the software;
5. Re-test the changed software.

This thesis is concerned with the ease with which step 4 can be performed[1], which is determined to a large extent by the **evolveability**[2] of the software. Evolveability is often viewed as a qualitative software measure and, although quantitative complexity measures exist which aim to quantify maintainability and evolveability [McCabe76a] [Halstead77a], proof of their worth for representing maintainability is questionable. In addition, their ability to model exactly the evolveability of software is questionable because of the concentration on product issues and the consequential lack of modelling of environmental factors (however, see [Bennett98a] for a more comprehensive maintainability model). The primary purpose of this thesis is to improve the evolveability of software by improving the **flexibility**, **adaptability**, **extensibility** and **localisation of evolution** of software through design for change. There are three broad prongs to the thesis:

- Identification of a set of **software entities**[3] with increased evolveability and tolerance to change. A software entity is an abstraction or construct used by software engineers in developing software. Examples include tasks, services,

[1] It should be noted that the thesis doesn't help with describing *how* a new requirement can be expressed in terms of the software entities identified in chapter 5. This is a task well-suited to humans.

[2] Boldfaced terms in the main text identify terms which may either have an ambiguous definition or have been introduced as new terms in this thesis. They are defined in chapter 2 table 1.

[3] A software entity is an abstraction.

data conversions, objects and messages. See chapter 5 for a more detailed description. A by-product of the increased number of software entities (or abstractions) is improved localisation of software evolution;

- **Increased evolveability of software entities encompasses:**
  - **Increased flexibility**; this expresses how easy it is to change a software system in response to changes in requirements. This is linked to the constraints (such as assumptions, design decisions etc.) imposed on the software entity – the more constraints, the less flexible;
  - **Increased extensibility**;
  - **Increased adaptability**[4]; this expresses how well an *existing* software entity adapts to changes in its environment, for example in response to changes in software entities on which the software entity depends;

  The terms "adaptability" and "flexibility" are usually inter-changeable [Thompson92a]. However, for the purposes of this thesis, they are used to express different forms of adaptability: adaptability represents adaptability of individual software entities with respect to changes in other software entities on which they depend, and is concerned with **secondary evolution**. Flexibility represents the adaptability of a software system as a whole, and is concerned with **primary evolution** *and* **secondary evolution**;
- The use of open implementation techniques, where possible, to re-cast **integration evolution**[5] as **re-configuration evolution**[6].

The overall approach is based on software entities adapting to changes in entities (software, hardware or other external entities such as requirements) on which they depend, and the thesis identifies methods of improving the adaptability of software. Ideally, requirements, design decisions, user desires and other aspects of a software system's external environment that affects the evolution of the software system should be modelled as entities and linked to other software entities in the software system which implement them, much as Karakostas advocates with his Teleological Maintenance model [Karakostas90a]. By doing this, changes in these external influences can be linked to the dependent implementation-level software entities. However, the complexity of the real world coupled with the lack of techniques and methods for relating complex and abstract requirements and user desires to software entities prevents this from happening at present. This leads on to two possible interpretations of "flexibility" in a software context:

1. Inherent flexibility of the whole software with respect to changes in requirements or other external influences, such as user desires;
2. Inherent flexibility of *existing* software entities with respect to changes in other elements of the software on which they depend.

The first is a difficult problem because new requirements will often conflict with existing requirements realised in the software. Also, an initially bad design or a design that is incorrect in the face of new requirements that invalidate the

---

[4] Adaptability, as used in this thesis, does not refer to adaptation in a learning context as it's used in artificial intelligence research.

[5] See chapter 2 table 1.

[6] See chapter 2 table 1.

design, is hard to change. A consequence of this observation is that software can't be *completely* inherently flexible with respect to new requirements, because new requirements may involve invalidating existing requirements and assumptions, which in turn require changing *existing* aspects of the software. No matter which way the real world is modelled, changes will inevitably occur, and some of these changes will conflict with the existing models. However, part of the hypothesis of the thesis is that some inherent flexibility with respect to requirements changes can be met. The second interpretation is easier to tackle because certain assumptions made by software entities with respect to other software entities on which they depend can be removed in order to make the software entity more adaptable. Another hypothesis of this thesis is that software architectures and models can be modified to improve a certain aspect of software evolution, that of how changes to software affect other parts of the software i.e. the flexibility of software, in the sense of the second interpretation above, can be improved. These software architectures can also be improved to increase the flexibility of software entities in the sense of the first interpretation above. It is the main aim of this thesis to show how this can be accomplished.

The difference between flexibility and adaptability, as used in this thesis, is that flexibility is concerned with how easy it is to change a software entity in response to a change in the requirements which **clients** of the software entity make of it. In comparison, adaptability is concerned with how easy it is for a software entity to overcome changes in those software entities on which it depends. Hence, adaptability deals with the range of changes in a server which don't require changes in the software entity, whilst flexibility deals with how easy it is to make changes to a software entity if adaptability fails.

In addition, the adaptability of a software entity is the level to which it can cope with changes to other software entities on which it depends. More specifically, a higher level of adaptability implies an increased tolerance to change in a dependant software entity. This is related to ripple effects (which are caused by changes in interfaces impacting on those aspects of the software which depend on the interface), since it is the dependencies between abstractions coupled with changes in abstraction interfaces that causes ripple effects.

The adaptability aspect of the thesis is tackled by identifying a set of software entities and the types of assumptions that they make about other software entities on which they depend. So, for example, function software entities assume a particular structure for the data software entities that they use. Adaptability is improved both by increased modularity (so that changes are hidden behind interfaces) and the extraction of assumptions and design decisions from software into parameters, where possible. Hursch observes the importance of identifying:

1. *What* can change in software. The parts of the software that are targets of change, which are called software entities in this thesis;
2. *How* the software entities identified can change [Hursch95a].

In general, the potential areas of evolution within a software system will not be known a priori, even in well-understood domains. Admittedly, there is a better chance when the software is well understood, and in this case measures may be built in to ensure that it is easier to change those parts more prone to evolve. However, in general, changes can't be predicted. In this thesis, an application domain independent approach has been taken to the identification of the targets of evolution, the software entities. For example, data adaptations, service adaptations and data-conversion adaptations are

particular classes of change for which measures are built into the software that allow such changes to be carried out. Additionally, they're domain independent.

The identification of targets of change (hereafter termed software entities) aids in identifying the types of change that can occur, which collectively prescribe *how* the software can evolve. A set of **software entities** that are targets for evolution is produced, which is coupled with the idea of an adaptation space [Cazzola97a] applied to each software entity which defines *how* the software entity can evolve. Hence, the adaptation space of a software entity prescribes what is possible and what isn't possible in terms of evolution for that software entity. The assumption is that the adaptation space of a software entity is determined by the adaptation spaces of the software entities making it up. Some software entities are primitive and cannot evolve (examples are algorithms that perform a well-defined task). The identification of types of adaptation of software entities means that measures can then be built into software to allow these types of adaptation to occur more easily, because they are more directly expressible within the **conceptual framework**[7] than existing software architectures allow. The main assumption here is that all future new requirements will be expressible in terms of these adaptation types and so will be amenable to the measures that have already been provided. This is related to the completeness of the identified change types and software entities and the notion of domain stability, a point to which the thesis returns in chapter 2 section 5.1.

The thesis makes considerable use of **reflection** (self-modelling) to help improve the specific area of software evolution research discussed in the previous paragraphs. The argument is that if more effort can be made in the software development stages, then this effort can be amortised over the course of the life of the software by easing future software evolution. Hence, increased modelling and use of reflection can ease future software evolution, although more effort must be expended in software development[8]. Reflection, as utilised in this thesis, essentially allows self-documentation of code by linking particular aspects of documentation that would normally be separate from the software to the software entities - an idea adapted from work by Karakostas on teleological software maintenance [Karakostas90a]. These aspects include:

- Dependencies or relationships between software entities;
- The types of dependencies between software entities.;
- How software entities can evolve (through the use of **evolution operators**) and their limits of evolution, which are determined by factors that are dependent on the software entity and ensure that a software entity doesn't break constraints imposed on it by its environment. These constraints may be that evolution must not break the behaviour of a **service**, or that the evolution of a **DEM** does not specialise the data in any way.

There are other factors that contribute to the ease with which changes can be made to software. A primary factor is the relationship between new requirements and the existing capabilities of the software. There are two aspects to this:

---

[7] "Conceptual Framework", in this context, means a set of software entities and relationships between these entities, with theory on how these software entities evolve and the effects of this evolution on each software entity's environment.

[8] This is similar to the argument in favour of software reuse.

1. The level of conflict between the new requirement and the existing assumptions and design decisions built into the software;
2. The overlap between the existing capabilities and the total number of capabilities needed by the new requirement. If a lot of the capabilities already exist, then relatively little work has to be done to evolve the software.

The second aspect leads on to the identification of two types of evolution:

- **Re-configuration evolution**, where the new requirement is expressible in terms of existing software entities;
- **Adaptation** or **integration evolution**, where new software entities are required that must be integrated with the existing software entities. This can be viewed as either an adaptation problem where the existing software entities must be adapted in order to satisfy the new requirement, or as an integration problem where the new software entities must be integrated with existing software entities. The choice is dependent on whether the viewpoint is the existing software entities or the new software entities.

Re-configuration evolution means that the capabilities required to satisfy a new requirement are already present in the software, so that evolution consists of changes in existing *instances* of software entities, be they changes in parameters or implementations of tasks.

A potentially useful classification can be made for software evolution, that of **primary evolution** and **secondary evolution**. Primary evolution is the primary change occurring as a result of changes in requirements. Secondary evolution consists of changes that occur as a result of primary changes invalidating assumptions. Improving evolveability aims to improve both primary and secondary evolution.

The main conclusion of the thesis is that complete inherent flexibility is not attainable, so that partial inherent flexibility must be settled for. The results of the thesis consist of a set of software entities with the following characteristics:

- Increased inherent flexibility to change, over existing software architectures and models;
- Information on how a software entity that isn't inherently flexible, can be transformed in order to adapt it to particular types of change.

It should be noted that the approach taken in this thesis is a domain-independent approach since it uses a set of domain-independent software entities and associated adaptations. These adaptations are also domain independent. The advantage of using a domain independent approach is that the results are generic. The main disadvantage is that there may be problems in modelling the domain in terms of the domain independent software entities, as opposed to specialised domain-specific software entities. These domain independent software entities also have the characteristic of being quite abstract. This seems the best approach considering that change prediction is a difficult task, which is eased the higher the level of abstraction that is used. For example, it is easier to predict that a new element may be added to a data structure than it is to predict that a particular element will be added to the data structure. However, the use of a more abstract approach such as advocated here means that it is more difficult to predict *how* a particular change will occur. This is true because an abstract software entity can evolve in many more ways than a more concrete software entity.

Most previous research on software maintenance and evolution has focussed on maintaining and evolving *existing* software systems. The approach described in this thesis depends on the assumption that the evolution is being performed on code that conforms to the conceptual framework developed in the thesis.

1 The Aims of the Research

Lloyd Osborn, in [Osborn93a], states:

> "Among the important features required of a system to ensure a fruitful life
> for it are: Adaptability, Flexibility, Scalability, Maintainability, Reliability".

The literature provides no consensus on what these terms mean, opting for qualitative descriptions that prevent comparisons from being made between different models and software architectures. Also, the classification is flawed, since maintainability is dependent on adaptability and flexibility. However, these features, or characteristics, are important for the purposes of software evolution and current software architectures and models fair quite badly with respect to them.

The main aim of this thesis is to ease the making of changes to software by improving the evolveability of software, by:

- Increasing the flexibility and adaptability of software, by:
  - Limiting the assumptions that software entities make about the software entities on which they depend;
  - Localising evolution within software entities by improving modularity.

Modelling is recognised as an important aspect of any software, be it object-oriented, functional or agent-based because all software is essentially a model of the real world that it automates. When writing software, software engineers need to make design decisions about how to model these concepts. However, these models are typically insufficient when it comes to evolution because they are essentially development-oriented models. Whilst increased abstraction has been rightly argued as a means of encapsulating changing aspects of the code, all too often abstractions are chosen for other reasons, so that software evolution suffers. The inescapable engineered characteristic of modern software results in the real world being modelled as a set of related abstractions (or software entities) that often conflict with the requirements that software evolution places on them. This thesis concentrates on:

- *How* changes are made to software entities;
- The effects of changes to software entities on other parts of the software;

and develops a conceptual framework (described in chapter 4), which provides a basis for the development in later chapters of theory on how flexibility and adaptability can be improved, and on how improved localisation of evolution is achieved.

To summarise, the aims of this research are:

1. The development of a conceptual framework for software evolution and flexibility;
2. The determination of the types of model and information contained in these models that are required to allow software to be more robust with respect to evolution and flexible when evolution occurs;
3. The models utilised don't depend on the software engineers having to predict what may change in the *requirements* of the software, but may have predictive elements of a more abstract nature (e.g. a data structure can change by the addition of a new attribute. The point is that changes to a specific data structure, for example, aren't predicted, only changes of a more abstract or generic nature, for which measures are built into the software to allow these changes to be made more easily). Reflection, coupled with a set of software entities produced using a higher level of modularization (see chapters 4 and 5), allows the software engineer to provide an evolution interface to these software entities. This evolution interface explicitly describes how these software entities can evolve in a particular context. For example, functional software entities can evolve with respect to functional behaviour (a sort program can have many different implementations that provide the same behaviour, but with different non-functional characteristics);
4. To provide a better understanding of product issues of software evolution i.e. how the product influences the ease of software evolution. It is hoped that the thesis provides a better understanding of how software evolution works, by giving some insight into inherent characteristics of software evolution which cause problems that can't be overcome, and by providing a better understanding of flexibility and adaptability, which are two important aspects of software evolveability.

The latter point about not having to predict future requirements is an important one. Software that was constructed using such an approach would probably be easier to evolve (and hence software evolution would only be re-configuration evolution) because the software engineers had foreseen the change and built in measures to ease evolution, but this approach can't hope to predict *all* future requirements. For example, in a 2-D graph model, it would be useful to have a constraint that invokes functional evolution in the graph layout algorithm when a new co-ordinate is added. This would allow data evolution to trigger functional evolution but depends on predicting that a new co-ordinate may be added. In summary, measures built into the models of the software to ease evolution do not depend on any predictive capability on the part of the software engineers. Any predictive elements will be confined to abstract, domain-independent software entities and not specific instances of these entities such as a particular data structure.

2 Assumptions

- Software is only as good as its models. For traditional non-reflective code, that means the software is only as good as its code. For reflective software, this also extends to the built-in models which are a part of the software. Software engineering currently produces non-reflective code in which documentation is separate from the code itself. Changes are hard to make precisely because the code is passive when it comes to evolution; it doesn't know what the parts that make it up do, how these parts can change, or how changing one of these parts affects other parts;

- The work described in this thesis depends on software that is qualitatively different from traditional software; specifically, software constructed using the principles expounded in this thesis and which conforms to the conceptual framework described in the thesis;
- The theory in this thesis assumes no prediction of *specific* future changes (in requirements) for the software system. Specific evolution changes cannot, in general, be predicted. Previous projects in a domain(s) may help in this regard but cannot in general be relied upon. A related assumption made, however, is that, although specific changes can't be predicted, more abstract/general changes can be predicted, and specific measures built into the software to enable changes of this type to occur more easily;
- Ripple effects occur because of changes in assumptions that exist between two dependent software entities. A reflective model that describes all dependencies should then be able to provide a way of identifying when ripple effects occur, and how a dependent software entity can be adapted to specific types of change in a software entity on which it depends;
- Software entities adapt in well-defined ways to changes in other software entities on which they depend.

3 Research Method

The research method adopted in this research has been an engineering one based upon an iterative process of improvement of existing software constructs with respect to evolveability; specifically, trying out or implementing a new construct, evaluating the result with respect to flexibility, learning and improving the construct. This approach, even though it is abstract and gives no precise guidelines for the various stages, was felt to be best suited to the kind of work being performed, that of improving existing software architectures with respect to flexibility. The characteristics of this work are based heavily on determining the best constructs to use in software, which led naturally into an engineering approach. However, with this type of research method, the evaluation is all important in guiding the research to the required solution. The evaluation needs to be as detailed as possible. The overall goal of increased evolveability is not very quantitative and therefore not very useful in this regard. Hence, the following section on evaluation is fairly detailed.

4 Contribution of Thesis

The main contribution to software evolution research is an improved understanding of software evolveability; what it is, the characteristics of a software language, architecture or model which affect it, how evolveability can be improved. This is achieved through the development of a conceptual framework or architecture for improved software evolveability, called SEVEN (for Software EVolution ENvironment), that comprises a reflective model consisting of a set of software entities with improved evolveability with respect to existing software languages, architectures and models. The software entities are chosen for their improved evolveability and, in addition, provides the software engineer with the following information:

- How software entities can evolve;

- The dependencies between software entities and the assumptions that client software entities make of the server software entities on which they depend. These assumptions constitute the interface of the server, which forms a contract between the client and the server;
- The effects of changes in software entities on their interface, and the effects of these changes in interface on clients of the software entity.

This information can be used to determine the effects of changes in a software entity on other software entities, but doesn't claim to completely model all assumptions made. The architecture does, however, provide an architectural harness that allows the effects of changes in a software entity to be mapped to effects on its interface and, in turn, how these effects on the interface can be related to ripple effect types.

5 Evaluation Criteria

The primary objective of this work is to increase the evolveability of software. The process of achieving this goal has been an engineering one, consisting of gradual improvement of software entity evolveability through case studies. The evaluation must show an improvement of the proposed approach, SEvEn, with respect to existing software models. The evaluation in chapter 8 is thus based on a comparison of SEvEn against existing software models with respect to the following criteria:

1. The level of improvement in flexibility;
2. The level of improvement in localisation of software evolution;
3. The level of improvement in adaptability;
4. The level of improvement in extensibility;
5. The level of improvement in the detection of ripple effects;
6. Support for **re-configuration**, **integration**, **primary** and **secondary** evolution. How well the proposed model aids these types of evolution;
7. How the work contributes to a better understanding of software evolution and evolveability, and how this affects software evolution.

6 Thesis Structure

The thesis first describes why software evolution occurs and why it currently costs so much in terms of time and money (chapter 2). Chapter 3 describes candidate approaches to software evolution. Chapters 4 and 5 introduce and describe a conceptual framework for software improved software evolveability. Chapters 6 and 7 describe data and functional evolveability, respectively. Chapter 8 provides an analysis of the evolveability of the conceptual framework by evaluating it against the evaluation criteria in 5 and against current software languages, models and architectures. Chapter 9 concludes the thesis and discusses directions for future research.

Chapter 2

What is Software Evolution?

1 Introduction

This chapter describes the broad context of the research, that of software evolution in general. Later chapters concentrate on the specific software evolution research areas of software evolveability, flexibility, adaptability and extensibility. The chapter describes what software evolution is, how it is performed, and what the problems are. Since there is no standard (IEEE or otherwise) definition of software evolution, the following definition clarifies the term "software evolution" for use in this thesis.

Definition: software evolution is the set of processes and techniques used in changing software functionality to meet new user requirements which may revoke, extend, re-implement or conflict with existing requirements.

The definition assumes that so-called non-functional requirements can be expressed as a set of functional requirements, as Bass et al observe [Bass98a].

The processes mentioned in the definition provide a common pattern of changing software i.e. how they make changes to the software. Typically, these processes are based on a change-request model that is very abstract and doesn't provide much help in identifying *how* to make changes to the software. Hursch identifies four important questions regarding the change-making aspect of software evolution (excluding testing which, though important, is not of concern to this thesis):

1. Who changes the software?
2. What can be changed in the software?
3. How are the software elements identified in (2) changed?
4. When are these software elements changed? [Hursch95a p23].

Point one is important because how evolution is performed should be targeted to who is making the change. For example, end users should be able to perform simple types of evolution that involve re-configuration i.e. changing parameters such as what colour a window is. Software engineers should have access to more powerful forms of evolution based on integration where the change involves the addition of new capabilities. Point 2 above relates to the observation that any new requirement must be expressible in terms of the basic constructs of the modelling framework provided by the software languages used today, such as data structures, functions, conditionals, loops etc. How these can be changed constitutes how a software system as a whole can be changed.

In addition to these, there are two other important questions that help us gain an understanding of what software evolution is. These are:

5. Why does software evolution occur?
6. How can software evolution be performed?

Point 5 is explored in the remainder of this chapter.

Before exploring point 6, another definition is required.

Definition: The **configuration** of a software system describes the software entities in a particular software system and their relationships to each other.

Point 6 can be subdivided into three categories which contribute to the change-request process of software evolution [Bennett91a], and which consists of:

- Identifying the need for change;
- Constructing a requirement that encapsulates the required change and creating a change request for the new requirement;
- Mapping this change request requirement to changes in the software. This brings us to the identification of two types of evolution:
  - **Re-configuration evolution**, in which the change can be expressed in terms of the *extension* of the software, that is in terms of changes to particular parameters. Hence, re-configuration evolution is also termed "parameter evolution". A major advantage of re-configuration evolution is that ripple effects shouldn't occur, because no interfaces and no assumptions are changing. There are a couple of exceptions to this. Firstly, when changes fall outside the extension of a particular entity but aren't covered by the typing mechanism. As a simple example, consider a function that has an integer parameter, P. The extension of this parameter consists of all possible values that the integer type will allow, which may be outside the range of values that the function is capable of processing. Hence, a re-configuration change has been converted into an integration change because it requires an adaptation of the function so that it can process the larger extension of P. The second exception is essentially a specialisation of the first and concerns so-called non-functional behaviour, such as speed. The important observation is that such behaviour does not form part of a software entity's interface, so it is difficult to determine when a change in behaviour has occurred because there is no change in an interface;
  - **Integration evolution**, in which the change is performed by integrating a new set of **software entities** with the existing software entities of the software system;
- Determining how the changes made effect the existing software system, stemming from the fact that built-in assumptions, design decisions and requirements that conflict with the new requirements result in ripple effects.

The ease of mapping a new requirement to a change in the software system is dependent on the existing configuration of the software.

The following aspects are involved in the evolution of software:

1. A new requirement;
2. A set of software entities (see chapter 5);
3. A set of specialisations and instances of the software entities in 2 that form the software system's current configuration;
4. The evolution spaces of the software entities in 3;
5. A set of new software entities that have to be integrated with the existing software entities.

These five together designate the type of evolution that will occur, whether it be re-configuration evolution or integration evolution, in order to satisfy the new requirement. For example, given the requirement of changing from a bubble sort to a quick sort when a quick sort software entity doesn't exist, requires new software development followed by integration evolution. If quick sort does exist, then a re-configuration approach can be utilised. Given the new requirement of adding the time to the task bar of a window manager's desktop and a particular set of software entities that specifically doesn't include the functional capability to display the time on the desktop, will require an integration type change. If such a capability exists, then the change will still be integration evolution because the existing capability must be integrated into the control flow of the software.

Evolution typically proceeds in either a bottom-up or top-down manner by either re-configuring existing software entities or integrating new software entities into the existing software system. After each re-configuration or integration, the software system consistency has to be checked by performing ripple effect adaptations. So evolution consists of a sequence of changes, each consisting of a decision based on the following:

- How the existing software entities need to change to satisfy the new requirement, followed by consistency management;
- What new software entities are required in order to satisfy the new requirement, expressible in terms of existing software entities (integration).

Each change will determine whether the following integration will proceed bottom-up or top-down, because of software entities that are used by the change that need to be integrated with the existing software. For example, a change to a sort program's control flow so that a call to a bubble sort procedure is replaced by a quick sort procedure when a quick sort procedure doesn't exist, will have to result in the introduction of a quick sort procedure into the system. If performed immediately, this change will proceed top-down since one is moving down the abstraction levels.

Hursch approaches the problem of consistency management in the context of object-oriented software by studying how the software entities in an object-oriented system (the classes, objects and relationships) evolve and determining how changes in the class model affect both the instance (or object) model and the functional model [Hursch95a]. Adaptations are limited to fairly simple class transformations. His work is also constrained to object-oriented software, so that evolution in his framework is therefore constrained to classes and inheritance and reference relationships. Higher-level change targets such as software architecture aren't considered. There is no theory on changing the behaviour (the

methods, propagation patterns etc.) of the software because the emphasis is on how the software can maintain behavioural and structural consistency in the face of changes to the class model (or schema). His approach is basically a method of coping with ripple effects that result from changes to the class schema. This thesis aims to address other forms of adaptability and to develop a clearer understanding of the nature of software evolution, aside from evolution process issues.

Term	Definition
Adaptability	A measure of the range of changes in a server software entity for which a client software doesn't need to be adapted. Low adaptability of a software entity means that changes in those software entities on which it depends will often require changes in the software entity itself. High adaptability, on the other hand, means that the software entity deal well with changes in those software entities on which it depends.
Client or Dependant	A client of a software entity uses the software entity to do something, and is therefore dependent on that software entity's interface.
Computer System	Consists of a set of inter-related entities that compose a system. This includes the software system, users, hardware and any other entities that depend on or are dependent on the software system.
Configuration	A set of software entities and relationships connecting the software entities.
DEM	Data Entity Model, a meta-model describing the relationships between the data entities that make up a particular data structure.
DIM	Data Instance Model, an instance of a DEM.
Entity	An independently-modelled aspect of the computer system, including all the software entities in the software system, in addition to aspects such as: • Users; • Requirements; • Desires of users that result in requirement changes; • System administrators.
Environment (of a software entity)	The set of software entities that use and are used by a software entity.
Evolution Operator	Defines an evolution operation on a software entity. A set of evolution operators for a particular software entity defines the evolution space of that software entity. An evolution operator is applied to a source configuration to produce a target configuration. Evolution operators typically consist of add and remove, applied operations to a set of software entities which form the configuration.
Evolution Space	A set of configurations "reachable" from the current configuration, using a set of evolution operators, and satisfying the constraints on the configuration. For example, the evolution space of a software architecture is determined by its evolution operators add/remove component/connector and any constraints on the evolution, such as "The target configuration

	must have a star topology". The evolution space of a service is determined by the evolution operators add/remove message and the constraint that the target configuration still satisfies the requirements of the task it implements.
Evolveability	A measure of the level of resistance to change (in general terms i.e. for all future requirements) of a software system. Includes measures of adaptability, flexibility, extensibility, and localisation of evolution.
Extensibility	An aspect of evolveability; a measure of the support provided by the software model for the dynamic addition of software entities.
Flexibility	An aspect of evolveability; a measure of the level of resistance to change of a software system as a whole with respect to changes in requirements.
FSE	Functional Software Entity, an abstraction used to model functional aspects of a problem domain.
Integration Evolution	Evolution is concerned with the introduction of new capabilities (functional, data or otherwise) to a software system in the form of new software entities.
Interface	Every software entity possesses an interface, which is dependent on the software entity. For example, the interface of a service is the set of formal parameters, plus behaviour, speed of execution and memory requirements. The interface of a DEM is the set of data entities and relationships in the DEM.
Non-primitive Software Entity	A non-primitive software entity is prone to evolution, defined by its evolution space operators.
Primary Evolution	This is evolution which occurs as a direct result of a change in requirements.
Primitive Software Entity	A primitive software entity is a software entity which doesn't evolve. The reason for this is dependant on the type of software entity. For example, an algorithm is a primitive FSE. A stable DEM is primitive and doesn't change because it models a domain well.
Reconfiguration Evolution	Evolution which is concerned only with changes to *existing* instances in a software system. Specifically, no new capabilities are required.
Reflection	The ability of a software system to model aspects of itself, which the software can then manipulate.
Secondary Evolution	This is evolution which occurs as a result of conflicts in requirements, which results in ripple effects.
Server	A server software entity is used by another software entity, and performs something on its behalf. For example, a particular FSE requests another server FSE to perform a task on its behalf.
Software Entity	A domain-independent abstraction used in the modelling of the real world to create a software system. Software entities are a type of entity.
Software Model	The term "software model" is a synonym for software language, architecture or model.

Table 1 – Thesis Terminology

Figure 1 shows how a client-server relationship exists between a software entity that depends on or uses another software entity[1]. Most software entities in a software system will take the role of both client and server. For example, a function can use other functions and be used by other functions. Functions are an interesting example because they are mutually dependent. If function A uses function B, then A depends on B's interface (formal parameters) and B depends on the actual parameters A passes to it. Hence, A is both client and server and B is both client and server. The terms "adaptability" and "flexibility" can be defined in terms of this classification of software entities. Flexibility is defined as the ease with which a server can be changed in response to changes in a client (if the client is a requirement, then flexibility is "flexibility with respect to requirements"). Adaptability, in comparison, is defined as the ease with which a client can cope with changes in the servers it uses.

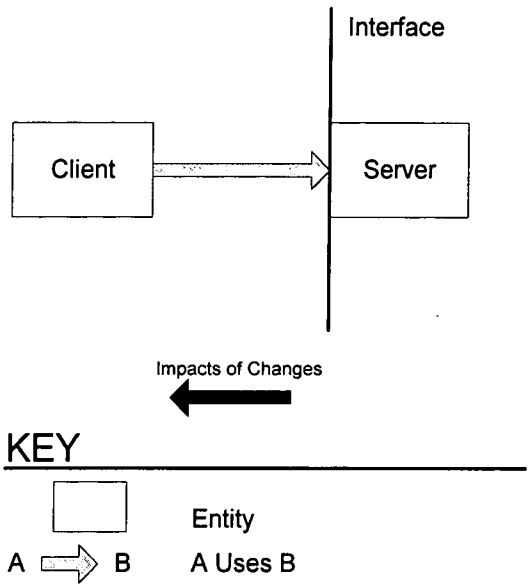


Figure 1 - Client and Server Entities

The following sections attempt to elaborate on some of the points above and provide a clearer context for the thesis.

2 Why Does Software Evolution Occur?

Maintenance, and more specifically evolution, is a fact of life for software systems, particularly those that automate some real-world task or activity. As Lehman's first "law" of program evolution states, a real-world software system must change or become progressively less and less useful to its owners [Lehman85a]. This is a direct result of the fact that such systems model some aspect of the real world which is itself changing, leading to changing requirements of the software system. In their SPE software classification, Lehman and Belady call this type of software system a "P-type" application [Lehman85a]. P-type software has validation concerns when deployed in a real-world situation; the system may match its requirements but the requirements may be incorrect or may evolve because the real-world application that

[1] This isn't to be confused with distributed object terminology where client-server systems have their own definitions of the terms "client" and "server". In this thesis, a "client" is any software entity that uses or depends on another software entity, and a "server" is any software entity that is used by another software entity. A software entity can take on either role, even at the same time.

they are modelling evolves. This has consequences for the software system, which must evolve too. "P-type" also encompasses those applications for which the specification is hard to define or there is no specification, either because the requirements are very abstract or there is no easy way to specify the requirements. Many AI applications fall into this category because they can't be specified easily (or at all); typical examples include chess playing software, neural networks and software with emergent behaviour [Forrest91a, Jantsch80a, Holland98a].

To complicate matters a further type of software system, the "E-type" application, extends the notion of a "P-type" application to include recursion; "E-type" applications are a part of the world they model, which means that they have unpredictable effects on their operational environment [Lehman85a]. These effects[2] cause the environment to evolve, which causes changes in the way it is modelled. This, in turn, has an effect on the system and so on. Additionally, the lack of understanding that can occur between a user and software engineer when trying to elicit software requirements creates a need for evolution. It is for these reasons that a software system must evolve throughout its lifetime and why software maintenance costs between two and four times that of software development, on average [Sommerville92a p538].

3 What is Software Evolution?

The previous section described some of the reasons why software evolution occurs. In essence, changes in a software system's environment trigger changes in the software system itself. This dependency can be extended to all levels of abstraction in a software system, from small components to entire subsystems, because all such software elements possess an environment. As described in chapter 5 section 2.1.2, a software entity's **environment** consists of all other software entities on which it is dependent, or which depend on it. By extending the use of software architecture to all levels of abstraction in a software system, one can conclude that there are basically two types of software entity in which changes can occur that will affect any other software entities in its environment, namely **components** and **connectors** (to use Garlan and Shaw's terminology [Garlan93a]). In other words, changes in component state and changes in messages sent between components provide the triggers for evolution in a component or connector[3]. What this evolution involves is dependent upon finding a finer classification of the types of evolution that can occur to these two types of software entity, which is in turn dependent upon finding a finer classification of these two types of software entity.

3.1 Software Evolution Triggers

Like any other successful product, successful software is self-sustaining because it builds up a community of users and other software products that depend on it and which continually require more and more capabilities of it. A software system is not an isolated product. It possesses an environment, which consists of two major types of entity[4]:

---

[2] For example changes in working practices as a direct or indirect result of the introduction of the software system.

[3] Of course, such changes are themselves triggered by influences outside the software system.

[4] An entity is a part of a software system (where a software system consists of software, hardware, users and anything else that may *directly* use or be *directly* used by any part of the software system).

- Entities that use the software system. Examples include users and other software systems;
- Entities that are used by the software system. Examples include the hardware on which the software system is executed, and software technology which the software system uses, for instance, CORBA or an operating system API.

Changes in both types of entity have the potential for affecting the software system. The real problems posed by software evolution, however, are traditionally influences in a software system's environment (such as users) for which the relationship is either difficult to determine or not modelled. This is typically the case for requirements (or user desires), which are only linked to software system entities through documentation. Hence, non-trivial changes to requirements are difficult to link to changes in the code because the relationships are not well understood. Hence, there are two types of trigger for software evolution:

1. Changes in entities/influences external to the *software system's* environment, such as users;
2. Changes in the (software) entities in the environment of those software entities which constitute the software system.

Type 1 triggers are difficult to handle because, unlike type 2 triggers:

- The mapping between them and the software is often quite complex. For example, the mapping between many non-functional or high-level requirements and those software entities which implement the requirements is often very complex. Consider the mapping between a speed or execution requirement and the software entities used to implement the requirement;
- They are not modelled in the software system and so it is difficult to determine the relationships between the influencing entity and the relevant aspects of the software system. Documentation helps in this regard, but it is still difficult to determine the precise relationships between the external influence and the aspects of the software system that it influences.

The level of abstraction is important for determining the environment and hence the triggers for evolution. Each entity of a software system, from the whole system itself down to individual functions and data structures, possesses an environment. Changes in the environment may or may not affect the entity. For example, a new requirement is a change in the environment of a software system, which will affect the software system if the requirement isn't implemented, but will not affect the software system if the requirement has been implemented. Similarly, a change in a function that breaks the interface of the function will affect any functions that use the changed function. But, not all changes in the function will break its interface.

The question of what is involved in software evolution can be made more specific by considering which aspects of a software system evolve and how they evolve. This can help in answering the question of how these individual aspects can be made to evolve more easily. This latter point is dependent upon the way the software is modularised, an important aspect of the thesis which is explored fully starting in chapter 4.

Type 2 triggers can be realised in one of two ways (with appropriate functional additions made to software entities to allow them to exhibit the required trigger behaviour):

- As a **reactive** mechanism, whereby specific changes in a software entity form the pre-condition of an action (a control-type message such as the execution of a function) which is executed when the pre-condition becomes true. For example, a particular change in a data structure can cause a particular message to be sent to a particular service;
- As a **responsive** mechanism, whereby an "interested" software entity periodically polls the state of another software entity to detect particular changes. In effect, the software entity registers its interest in a particular type of change in another software entity.

Ultimately, all software evolution is triggered by type 1 triggers i.e. changes in the software system's external environment. Some triggers are easier to link to the evolution that has to occur to re-stabilise the software, others are more difficult to link to the evolution. An example of the former is a change in the messages received by a server (not new messages), having the characteristic that the change occurs within an evolution space that allows the software to interpret it. Examples of the latter include changes in the desires of the software's stakeholders that should trigger evolution, but which are difficult to link to where the evolution should occur because they aren't modelled within the software. If such desires were modelled within the software, with an appropriate way to model changes within the software as well, then such changes could automatically trigger appropriate changes in the software, much like changes in one part of a software system will trigger changes in other parts. The important point is that, through reflection, the relationship between the source model (the desires or requirements) and the target model (the implementation of the desires) is explicitly modelled. This framework makes it simpler to model the effects of changes in the source model, assuming that the effects of changes in the source model on the relationship and target model are well understood. Like any other engineered product, however, such a framework would fall prey to changes outside of the provided interfaces i.e. changes that weren't expected when the framework was designed. This approach is the thrust of, for example, Karakostas' work [Karakostas90a].

A software entity's environment is therefore "connected" to the software entity through responsiveness and reactiveness, both of which involve the transfer of messages. Therefore, the assumption is that the source of all evolution will be in the messages that software entities receive[5]. By adopting the general message classification of responsiveness and reactiveness, all patterns of interaction of a software entity with its environment can be described. For example, responsiveness allows one to consider a functional software entity "polling" another (possibly non-software) entity such as a user of the software system, in order to determine a desire for change. This could be modelled as the functional

[5] This view can be defended by the following argument. Computers are essentially "dumb", requiring outside help in order to make them do something useful in the form of a software engineer producing code which the computer executes. Any evolution change in the software will be as a direct or indirect result of changes in the software system's external environment (for example, human users, hardware sensors, the passage of time triggering a task etc). Therefore, any evolution change can percolate through the software (producing evolution changes in the software as it goes, due to ripple effects) only through the messages that software entities exchange. Any changes in the internal data structures of a task will, for example, originate in an external influence.

software entity sending out a request to the user entity, with the response encapsulating the change in desires of that user entity. Typically, however, the links between environmental changes and changes to the software (termed evolution) are not known a priori, making it difficult to determine the effects of the change. The factors that influence the complexity of evolution are:

- The complexity of the dependencies between the software entities (or components) of the system (where the system includes the software system and its environment, which includes users, other software systems and hardware);
- The level of abstraction. A software entity's environment may be large or small, depending on the software entity. Similarly, the environment of a software system as a whole may be large or small.

Functional capabilities combined with rules that define how these functional capabilities can be composed into processes (or threads of control – see chapter 5 section 3.2) determines the behaviour of the software. Typically, the rules are provided by the software engineers constructing the software and, unlike the services, are not explicitly modelled. Both are potential targets of evolution; new functional capabilities, control elements and combination rules can be added or existing ones modified. Control messages are used by the control elements to invoke functional capabilities. Changes occurring in control messages, like in all other software entities, are of the following types:

- Changes in existing control messages;
- New control message type.

4 How is Software Evolution Performed?

Is there one approach that covers all types of evolution? The answer to this question will be easier to answer once one determines the relationship between a software system before and after evolution, the subject of the next section.

4.1 Relationship Between Software Before and After Evolution

There are two main types of relationship between a **target configuration** (the software entities in a *target* software system, that results from evolution to satisfy a new set of requirements) and a **source configuration** (the software entities in a *source* software system, before evolution has taken place):

- Re-configuration type relationships;
- Adaptation/integration type relationships, such as:
  - Specialisation i.e. software entity B specialises software entity A e.g. software entity "Sort" specialises software entity "Filter";
  - Generalisation, the opposite of specialisation;
  - Implementation i.e. software entity D implements software entity A e.g. software entity "BubbleSort" *Implements* software entity "Sort".

A re-configuration type relationship means using the existing software entities of the software system in a different way, for example the creation of a new process instance[6] to provide a new functional capability with the proviso that the services used by the process already exist. An integration type relationship, however, is arguably the most complex of the two since it requires the creation of new software entities either from scratch or by changing existing software entities, followed by integrating them with the existing software entities of the software, followed by the containment of ripple effects. These new software entities must be integrated with the existing software entities in a way that satisfies the change in requirements. Therefore, in order to qualify as a reconfiguration, a change must use only *existing* software entities.

Figure 2 shows the dependencies between a new configuration and an old configuration. (a) depicts a reconfiguration type change that can be realised (or expressed) in terms of existing software entities, making explicit what is currently only implicit in the software. (c) depicts an integration type change that can't be realised (or expressed) in terms of existing software entities. (b) depicts a combination of the previous two. Both (b) and (c) require new software entities. This is a recursive definition that will continue until (a) has been reached. In other words, a point will eventually be reached when all new software entities will be expressible in terms of only existing software entities. The route by which this is achieved, however, is not straightforward in non-trivial software. The process of making changes is essentially a planning process and the route isn't know a priori. Note that the previous description is basically a top-down approach.

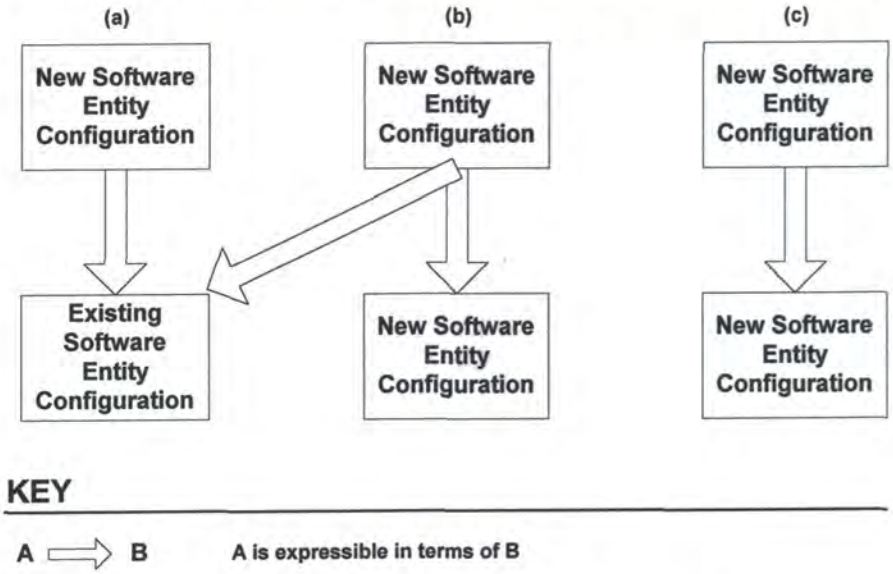


Figure 2 - Dependencies Between New and Old Configurations

As pointed out by Cazzola et al, there is a difference between reconfiguration and evolution in software [Cazzola97a p7]. Reconfiguration occurs within a space of potential configurations which are reachable from the current configuration using existing capabilities, whereas evolution involves either:

1. The introduction of new software entities, followed by integration with the existing software, or;
2. Integration with another system.

[6] A process instance is an instance of a process. A process is a process abstraction, a thread of control through a software system consisting of conditionals, loops and calls to services (see chapter 5 section 3.2).

This thesis views software evolution as the adaptation of a set of software entities in order to satisfy new requirements. 1 simplifies to 2 if the features are considered to be a new software system. In both cases, the assumption can be made that for integration to be possible there must be a set of concepts that are shared between the software systems to be integrated. In a sort program, the three separate functional software entities *input*, *sort* and *output* are integrated on the data structures they use since the software is essentially data-driven. They also share a common interface imposed by the filter software architecture, in the sense that all components of a filter software architecture receive data, filter the data and then return the data to the caller.

4.2 Two Types of Evolution

Evolution occurs to the targets of software evolution, the software entities. It is important to distinguish, as Hursch points out [Hursch95a], between changes that occur *within the space of behaviours* of the software and changes that occur *within the space of potential behaviours* of the software. The former are commonly referred to as extensional changes (for example, changes to the value of a variable during the execution of the software), and are characterised by the fact that the change is within the capabilities of the software and can be effected by a change in the parameters. The latter are referred to as intensional changes and result in software evolution. The extension of an FSE is the set of behaviours which the FSE can produce, depending on the values of its parameters, whilst its intension is the set of potential behaviours which the FSE can produce, limited by constraints imposed on it by its environment (hence, a sort function's intension is determined by behavioural constraints which impose constraints on the function's input-output relation). The behaviour of software is determined by its interaction with its environment, which determines the values of parameters and data structures. Software may encapsulate any number of behaviours, the one used being dependent on the interaction with the environment and hence the parameters chosen.

The use of dynamic typing complicates matters somewhat, because the intension of a variable is now determined both by:

- The set of types that the variable can take, which is itself determined by how the types are related (in an object-oriented model, for example, types are typically represented by classes and related through inheritance);
- The set of values for each of the types.

This extension/intension dichotomy can be extended to all software entities in the software. Evolution has to occur in terms of concepts already known to both the software and the software engineers who are making the change. Evolution operators are therefore essentially intension operators that describe all possible "values" of a software entity. The concept of relativity can be adopted i.e. intension operators are relative to the entity to which the intension operator applies. So, for example, evolution of a filter is evolution relative to that filter entity and the intension is the set of all filters, and evolution of a sort entity is evolution relative to that sort entity and the intension here is the set of all sort algorithms – bubble, quick, merge, heap etc.

For the example software entity model in Figure 3, extensional changes will consist of those shown in Table 2. Intensional changes will consist of those shown in Table 3.

Hursch's work defines software evolution as:

> "...the process of changing the schema [conceptual model], followed by changes to the objects and the methods to restore their consistency." [Hursch95a].

He doesn't partition evolution into types, but instead addresses all evolution in the same way. Making a change consists of changing a class schema. Integration type evolution is not addressed explicitly, even though it is an important type of evolution that deserves a separate approach than mere extension-type changes or reconfiguration type changes.

Operator Type	Operator Target
Add	<Instance> *HasA*[I] <Instance> relationship
Remove	<Instance> *HasA*[I] <Instance> relationship
Add	<Instance> *InstanceOf* <Entity> relationship
Remove	<Instance> *InstanceOf* <Entity> relationship
Change	<Instance>

Table 2 - Extensional Change Types

Operator Type	Operator Target
Add	<Entity> *IsA* <Entity> relationship
Remove	<Entity> *IsA* <Entity> relationship
Add	<Entity> *HasA* <Entity> relationship
Remove	<Entity> *HasA* <Entity> relationship
Change	<Entity>

Table 3 - Intensional Change Types

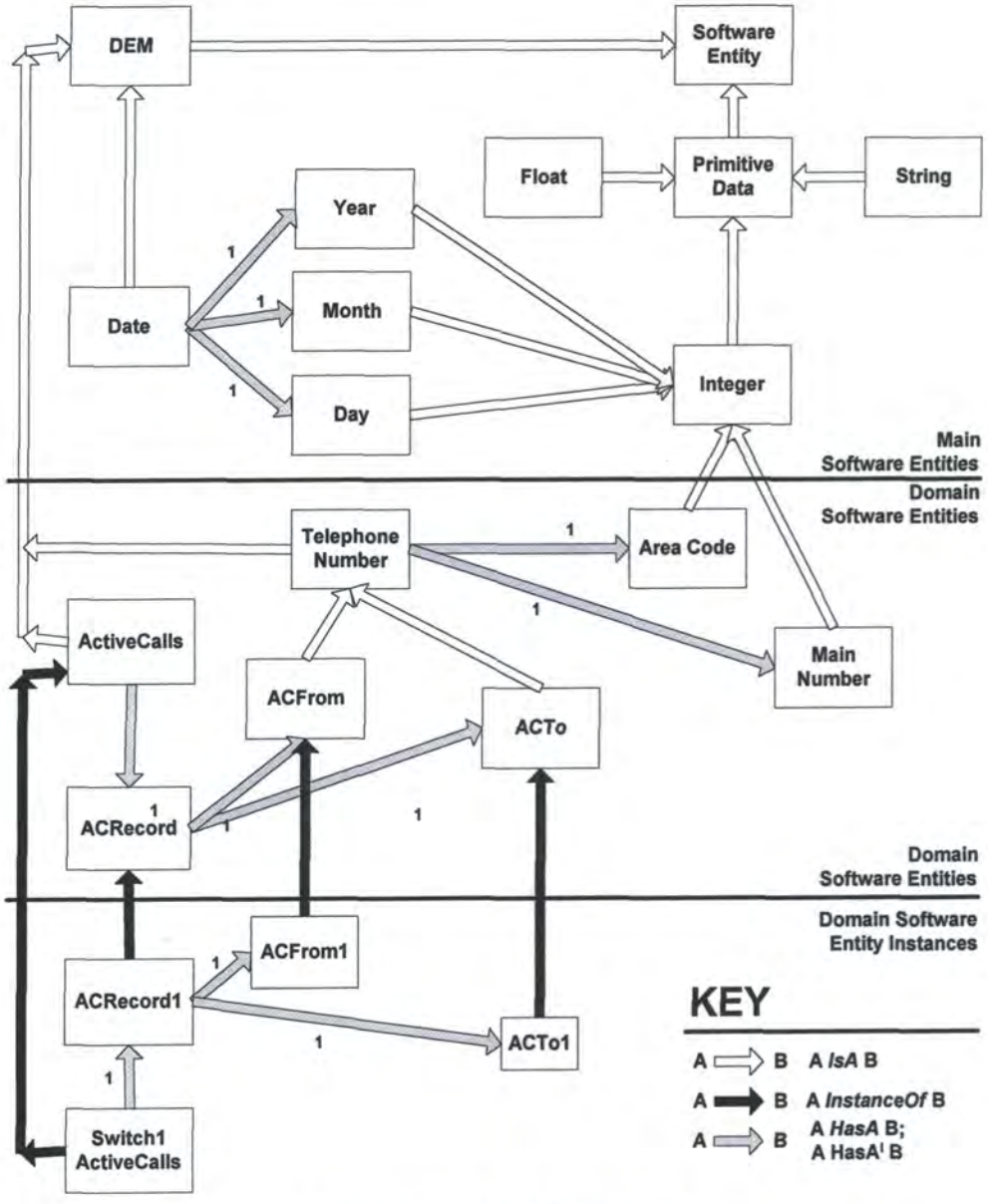


Figure 3 - Software Entity Model Example

4.3 The Process of Software Evolution

There exist many process models of software evolution. They are typically heavily based on software maintenance process models, which consist of three main phases, as identified by Boehm:

- Understanding the existing software (also known as **system comprehension**);
- Modification of the existing software;
- Revalidation of the modified software. [Boehm76a] [McDermid91a chapter 20] .

Each of these can be broken down into more detailed phases.

In large software systems, the whole process of evolution (and maintenance) is typically triggered by the submission of a **change request** to the **change control board**. A change request details the changes to be made to satisfy the new requirements, typically at a very high level of abstraction. An important aspect of this is the relationships and dependencies between the changes to be made to the software. Some changes will trigger others by causing changes in the environment of a software entity. Others will, by their implementation, form such triggers for change in other parts of the software. This has consequences for the order in which changes are made and thereby for system comprehension. This is the case because of temporal dependencies between the changes. For example, changes to the control aspects of software are difficult because control has a temporal characteristic so that particular services will be used before others. This means that making a change that executes immediately after another change before that other change has been made will have effects on the latter change by altering its interface. For example, imagine that change B executes after change A and change A is made first. This means that the environment of the entities affected by this change will also change, thereby triggering evolution in these interfaces. This is known as a **ripple effect**, and can be used to advantage to determine all the effects of a particular change in requirements. There is a problem, however, in determining the source change(s) that trigger all other required changes in the software (and thereby helping in program comprehension to determine what else needs to be changed in order to satisfy the new requirement).

It is difficult enough mapping a change in requirements to the set of changes that must be made to a software system in order to fulfil the change, but the real problems in software evolution lie in determining what must change and what doesn't change in response to a change in requirements. Ripple effects are unwanted side effects caused by the invalidation of in-built assumptions from previous requirements, which conflict with the new requirements. They have traditionally been solved passively, by ripple analysis and impact analysis techniques [Yau78a]. Another problem is ensuring the consistency of documentation with respect to code and consistency of documents with respect to each other. Lack of documentation consistency can be brought about by software maintainers failing to update documentation, or updating it incorrectly, either because of time constraints or the difficulty of the task. Hence, maintainers tend to trust only the code.

Current software evolution process models are necessarily abstract and not very detailed in the way they describe how to convert an abstract change request into an implementation. They prescribe only how abstract change requests move through the software evolution process, without detailing how these abstract change requests can be expressed in terms of more concrete constructs at the implementation level of the software. This is the case because the "semantic distance" between requirements (expressed in the domain) and the constructs of the implementation language is often large, making it difficult to prescribe a generic procedure for performing the required mapping. The only way this could be overcome would be to lessen the gap between the two models, an approach that is at the heart of research into domain specific languages [Ward94a].

A major question to be asked of this research is where it fits into the software evolution process and where in this process it can be applied usefully. Firstly, this research is not concerned with testing and validation issues. It *is* concerned with software that allows changes to be made more easily than present approaches allow. Hence, it is a product-improvement approach which helps making changes to software. It doesn't provide a mechanism for helping to map requirements to

code, but does produce a software architecture/model which increases the evolveability of software and thereby makes this task more manageable.

5 Why is Software Evolution Difficult?

Changes made to code have to be mirrored in design, analysis and testing documents, whilst keeping all forms of documentation consistent. Additionally, testing has to be carried out after every change. Teleological maintenance and transformational maintenance are two related approaches which provide a framework for linking aspects of each stage of the software development lifecycle. Teleological maintenance provides links between abstract real-world concepts and the software concepts that implement them. These links are semantic and are similar to the "implements" relationship discussed in chapter 5. However, the links only describe dependencies, allowing software engineers to determine *what* will be affected by a change in a particular abstract concept, not *how* it will be affected.

Large software systems generally consist of many interconnected application domains. Some of these domains may be more directly related to the real-world application than other domains. Domains such as user interfaces and networking sub-systems are typically not directly related to any real-world application domain; they provide a supportive but essential role to the real-world application domains. The existence of high coupling between these domains results in difficulties in understanding and changing the software, because it is difficult to ascertain where in the software certain domains exist.

A major problem with software evolution is the impact that existing code can have on new requirements, due to conflicts between existing requirements, assumptions and design decisions built into the existing code conflicting with the new requirements. In telecommunications, this problem is termed **feature interaction**: the assumptions built in to the existing feature(s) (or telecommunications services) conflict with the assumptions of the new feature. For example, a redirect service assumes that an un-answered telephone call is redirected to another telephone number, an assumption that conflicts with an answering service which assumes that un-answered telephone calls are passed on to the answering service, which records the caller. There are two approaches to solving the problem of feature interaction and, by generalisation, requirements conflicts:

- A preventative approach, based on the production of guidelines on how to model the domain so that such interactions are reduced or completely eradicated;
- Design software so that conflicting assumptions are permitted, but not allowed to conflict with each other. There are two aspects to this:
  - Determining the parts of the existing software which may conflict with future requirements, an approach that involves some predictability on the part of the designer. A related approach involves the use of a formal requirements modelling framework that provides a basis for determining the presence of inconsistencies [Finkelstein94a]. This is traditionally performed at the source code level, which is a difficult level to perform it at. The disadvantage of performing inconsistency checking using a requirements language (which may be a domain-specific language or a logic-based language) is that new requirements must be expressed using the same concepts and terminology as existing requirements in the requirements language;

- Finding a way of overcoming the conflict. Some form of priority system could be utilised, whereby the new requirement takes precedence over the existing code. The problem with this approach is that assumptions are all too often implicit and therefore only manifest themselves when the software doesn't do what it is expected to do. No mechanisms can be built in to avoid this, because this would imply that the software knew its expected results/output and was able to compare this with its actual results/output.

The first approach seems to be the most popular (see e.g. [Zibman95a]. Such an approach results in an architecture (one architecture out of the many possible within the domain) in which feature interaction is eased, and which can be reused in many applications within the domain. The disadvantage is the domain-specific nature of the approach, and the fact that changes may occur to the domain that produce feature interactions that weren't predicted.

The following sections outline some of the reasons why software evolution is difficult.

5.1 The Specification of New Requirements

As Smith has pointed out, requirements can't be deduced purely in terms of the environment, even though this may be an important part [Smith96a]. This, of course, depends on what is modelled as part of the environment of a software system. Requirements originate from human users of the software system, which are part of the environment of every software system. By modelling user desires and thereby changes in user requirements, software systems could potentially react to changes in these requirements. However, no software system models everything in its environment, which means that changes triggered within the system by changes in the environment which aren't modelled by the software system, must be made by a human user. The lack of integration between requirements and implementation constructs through a common model means that software engineers must provide the mapping between changes in requirements and changes in the implementation.

A domain language is expressed in terms of concepts that form a common understanding between the software engineer and the person who designed the domain specific language. But, what happens if a new requirement can't be specified in terms of the domain language? The conclusions are that either:

1. Part of the requirement may be expressible in terms of the domain concepts;
2. None of the requirement can be expressed in terms of the domain concepts.

For aspect two, changes will inevitably have to occur to the domain language because it can't possibly model the whole environment of any software system modelled using the language, leading to similar problems as with typical software languages. Hence, the advantages for evolution offered by domain languages, such as ease of expression of new requirements, are no longer available and we must resort to integration-type evolution.

Additionally, there is the question of abstraction in the change specification. In a typical change request driven software maintenance process, the change request that specifies the new requirement will be fairly abstract. The remaining stages of the process allow this abstract requirement to be refined into a more concrete specification after system

comprehension has determined whether new code needs to be written or existing code can be adapted, or both of these are required. In the context of this work, the initial specification would need to be fairly concrete in order for the software to be able to parse and interpret it, in order for the software to then determine what is required in terms of its own evolution. This is a catch-22 situation: in order for the software maintenance team to be able to specify what it wants of the software system, the specification needs to be fairly detailed. However, in order for the specification to be fairly detailed, the software maintenance team needs to be able to come up with a set of such detailed change request specifications from the initial abstract specification. This requires some knowledge of the software, in order to determine how each aspect of the initial abstract specification can be implemented:

- As a reconfiguration of the existing code, exemplified by the use of the existing software entities in a different way;
- As an adaptation of the existing code, exemplified by the use of new software entities that specialise and integrate with existing software entities[7]. This can also be viewed as an integration of the existing code with new code, where the target code is a combination of existing components and new components.

This task increases in difficulty as the abstractness of the initial specification increases, in particular the "abstraction distance" (the difference in abstraction between the constructs in two models) between the requirements model or language, and the underlying implementation model or language. A high "abstraction distance" may cause problems in representing the information in the requirements model in terms of the implementation model.

In addition, a new requirement must be described in terms of concepts that *both* the user and the software know about. More formally, these concepts must have a common semantics i.e. the semantics are the same for both user and software. As Arthur states:

> "...the maintainer must examine the change and fit it into the existing system requirements. If the change doesn't fit the existing system design, then perhaps it should be developed separately as a new system rather than implemented as a kludge in the existing one." [Arthur88a p120]

In other words, a new requirement must be expressible in terms of the underlying implementation model, otherwise the new requirement is essentially incompatible with the implementation.

5.2 (Lack of) Domain Stability

Domain stability is an important aspect of domain specific languages. An important assumption is that domains and requirements aren't predictable/stable. For example, consider a simple POTS (Plain Old Telephone System) domain consisting of two operations "connect" and "disconnect". One could argue that a possible change may be to put a user on hold but in general this can't be predicted. [Arango91b] have argued that once a domain is fully understood it is stable

[7] Every integration change can also be viewed as a specialisation or adaptation type change if the level of abstraction is increased. For example, the integration of a new function into a software system can be viewed as an adaptation or specialisation of the software architecture for the subsystem concerned.

and worth the effort expended in trying to model that domain so that it can be reused. This approach is fraught with danger since any useful application domain will always throw up new operations and features as a result of user demand. The telecommunications domain is a particular example of this, as evidenced by the constant introduction of new services and features by telecommunications companies like BT. However, the software engineer may be able to build in measures to allow certain classes of change to be made more easily. This introduces the problem of choosing the classes of change that cover all possible future changes to the software.

In non-stable domains such as the telecommunications domain, domain-specific languages are not a good approach because they hard-code a model of the domain in the language. When the domain changes, the language has to change. This is problematic because domain specific languages (indeed any software language) often require a lot of effort to produce. In addition, domain instability, in general, can't be overcome.

5.3 Abstractions and Interfaces

Software engineering, like any other engineering discipline, works with abstractions and relationships between these abstractions. Abstractions possess an interface which clients of the abstraction use to access the abstraction. In software engineering, abstractions include the following:

- Functional components: for example, procedures, functions, methods, sub-routines, control flow;
- Data components: for example, data structures, abstract data types, objects;
- Structural components: for example, architecture, objects.

The work of David Parnas in the 1970s identified the need to hide aspects of the software that are liable to change (such as design decisions) behind interfaces [Parnas72a], so that changes in the abstractions are hidden behind the interface and don't affect any clients that use the abstraction. This, however, is problematic for a number of reasons:

- The interface isn't guaranteed to provide the most general interface to the current abstraction and all future versions of the abstraction. Indeed, changes to the abstraction may break the interface;
- It may be difficult to determine what the relevant abstractions are that may change. In addition, different abstractions in a software system may provide the same capability, but are not represented by the same abstraction (hence, as shown in Figure 4, the equivalent abstractions A_{2i} can be represented as one encompassing abstraction A_{11} in order to gain the advantage of only having to change A_{11} in order to effect a change to A_{2i}). It would be difficult to change these *different* abstractions in the same way because there is no common abstraction that represents them all. A prime example of this is the decision to use two digits for the representation of dates and to not use a common abstraction for this representation, a decision which led to the Y2K problem. This means that, even though each of these abstractions is essentially the same, there is no common abstraction representing them all and hence each one needs to be changed separately.

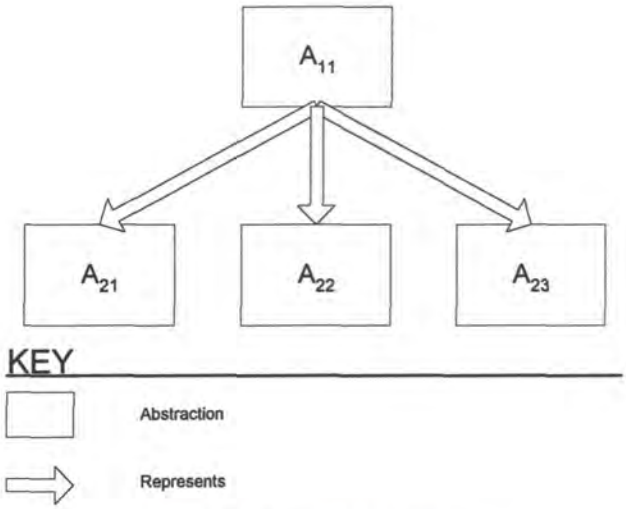


Figure 4 - Abstraction Relationships

5.3.1 Lack of Abstraction Generality

Abstractions used in the design of a software system are often too specific. This means that changes will invalidate the assumptions that these abstractions make. A solution is to make an abstraction as general as possible, a philosophy that is espoused in the UNIX operating system. An example of this is the changing of a software system to meet the user interface standards of another country, a change which involves translating on-screen text and the layout of on-screen entities. Such a change is typically difficult to make because of the sheer range of abstractions affected. This could be avoided by making a generalisation in the first place, hence removing the implicit country-dependence assumption. This, however, is heavily tied in with the predictability of changes, of which this is a particular example. Hence, generalisations imply the relaxation of the contexts in which the abstraction can be used (and possibly some predictability of future contexts), which in turn implies an increase in the generalisation of the interface to the abstraction.

5.4 Breaking of Requirements, Assumptions and Design Decisions

As section 5.3 discusses, software consists of a set of inter-dependent (or related) abstractions with in-built assumptions about other abstractions on which they depend. Software, by its engineered nature, also consists of design decisions. The dependence of aspects of a software system on design decisions causes ripple effects if these design decisions change and there doesn't exist an interface to shield these aspects from changes in design decisions. In addition, changes in an abstraction may invalidate the assumptions made by clients of the abstraction, causing ripple effects which may be quite far-reaching.

A naïve solution would be to encapsulate all design decisions behind an interface so that ripple effects are reduced. A potential approach could be to analyse the usage of concepts in a software system in order to determine how they are being used and, particularly, if the same concept is being used for different purposes. The fact that a concept is being used for different purposes means that the concept should be split into different concepts for each such use. The determination of this type of information is a difficult task in itself. However, even if it were possible, it is a characteristic of software that software evolution may expose un-encapsulated design decisions (and remove existing

encapsulated design decisions). For example, the design decision to use "Print" to display text messages on the computer screen doesn't cause any problems until evolution invalidates the design decision by breaking the implicit dependency between the requirement to display text messages and "Print", because "Print" no longer satisfies the requirement. Hence, the un-encapsulated design decision is exposed by evolution and causes ripple effects.

5.5 Context Dependence

Complete context independence is not possible in general because a lot of components are highly context dependent i.e. they only work in a small set of contexts. This leads to degrees of context dependence e.g. "redirect" has high context dependence, "sort" has low context dependence. Context independence assumes the reusability of components when components may not be reusable because they have a high context dependence, or coupling to a specific context.

A major problem with rule-based architectures like blackboards is that they can't be applied to all domains. One reason for this is the fact that pre-conditions and trigger conditions can't be used to model all functional units. Blackboards were essentially designed with data-driven domains in mind, so that control-driven applications are difficult to model using blackboards. Take, for example, a sort program that consists of *input*, *sort* and *output* components. To model these components using rules would involve providing them with a trigger and pre-condition each, which would essentially mean describing the environmental contexts in which they are executable. For highly context-independent components such as these, these trigger- and pre-conditions would be complicated and difficult to maintain, and would have to be specialised for each application. The *sort* component's trigger condition would be of the form:

"EventName = InputServiceFinished"

indicating that the sort component can be executed after the input component has finished executing. This is in opposition to traditional methods of coding, in which the main component would explicitly execute the input component followed by the sort component. So, in summary, the blackboard (or rule) based method seems very contrived and unnatural for control-based applications and domains.

A *sort* service has low context dependence on existing services. It can be developed separately. A redirect service for a telecommunications software system, however, has high context dependence on existing telecommunications services, specifically the implementation of the switch software. Such a service must be implemented in terms of these existing services, making software evolution more difficult because system comprehension techniques must be used to determine how to integrate the new service with the existing services.

The existence of context dependence will inevitably result in a phase of system comprehension, the level of which is determined by the level of context dependence; a high context dependence will result in a high level of system comprehension, whilst a low level of context dependence will result in a low level of system comprehension.

A problem with creating new capabilities[8] is that each new capability needs to be able to refer to other capabilities either existing in the software system or integrated into the software system through evolution. How can this be accomplished without the capability developer knowing how to incorporate the capabilities into the new capability i.e. knowing what the capabilities do and the API of the capabilities (how they're accessed)? For example, given a simple telephone switch consisting of connect and disconnect capabilities, how can an onhold capability be incorporated into the switch? Some knowledge of the capabilities required for onhold is needed. These capabilities may or may not exist in the switch software. Those that don't exist can be made available through evolution. System comprehension is inevitable because the evolution process inherently requires the software engineer to match the new capability (onhold) with the existing capabilities of the system. There is no way for the software engineer to write the onhold subsystem in a way that is detached from the underlying capabilities of the system because onhold is highly context dependent. He has to write it *in terms of* something that can be mapped onto the underlying software entities of the software system. This could be accomplished through a high-level domain-specific language or a low-level domain-specific language, or through some form of requirements to implementation transformation language.

The inherent context dependence present in software inevitably requires the software maintenance team first to comprehend the code (typically a time-consuming and costly task) before making the required changes. Additionally, documentation is traditionally separate from code, making it difficult to link documentation elements with their design and code elements. This means that information gleaned from the documentation may be hard to trace to the software design and code. This in turn means that maintainers tend to trust only the code.

5.6 Lack of Semantic Richness

Semantic richness, in its crudest sense, is defined by the number and range of constructs in the implementation model, such as constructs in a programming language. The range of constructs in existing programming languages is lacking to the extent that software evolution is not very localised. This, coupled with the lack of any structure to the relationships between constructs, means that:

- Changes are difficult to apply because the interface for the change isn't well-defined. This thesis proposes a set of software entities (or types) that provide both a set of constructs to build software, and a set of constructs with a well-defined evolution interface. This extended set of constructs provides a set of change types which provide an interface between the software engineer changing the software and the software itself, thus making the task of performing these types of change that much more straightforward;
- The effects of change are unpredictable because the internal structure of the software is not well-defined within the software code itself. This thesis proposes a reflective model that shows the relationships between software entities, that allows the prediction of how changes in one software entity can affect other software entities that depend on it;
- Changes that should be simple are made more difficult. This is because the targets of these types of change share characteristics, but this is either not immediately obvious from the code or is not modelled in the code. A classic

[8] A capability is a new element of a software system, such as a new service, a new data structure, a new architectural pattern etc.

example is a change that requires the start and end times of all function executions to be logged. The lack of any richness in the *IsA* hierarchy of functions (caused by the inability to change the compiler's underlying implementation of function execution) means that such a change affects every function in the code. If the model were richer, then a simple change further up the *IsA* hierarchy would ensure that all functions would be affected, as required. This is an advantage of many reflective models (for example, see [Maes87a, Maes87b]), which use an object-oriented approach to model the reflective aspects of the software.

5.7 Software Architecture Issues

Software architecture design is an important part of software development. Choosing the "best" architecture for a particular software system may help ease both development, evolution and other aspects of software development and maintenance. A given application domain may have "standard architectures", which are architectures that provide the "best" skeletal model for the particular domains concerned. For example, user interfaces generally have an event-based software architecture, and compilers typically share a common architectural style. There exist many different ways in which an architecture can be specified in different stages of the software development process. For example, informal diagrams, module interconnection languages, processes and frameworks [Garlan93a p1]. There is no clear mapping between the specifications in different stages of development and no means of formally specifying an architecture in order to aid reasoning. The creation of a standard schema that allows all software architectures to be specified in it would create a common ontology in which software developers could converse. Furthermore, the use of a standard schema would allow reuse of architectures. Currently, the design and specification of the architecture of a software system is mostly ad hoc, leading to informal specifications.

A software system has two main types of structure – abstraction levels and structure within abstraction levels, as shown in Figure 5. The former can be modelled using relationships that provide abstraction such as *Uses*, *HasA*, *IsA* etc. The latter can be modelled using software architecture theory.

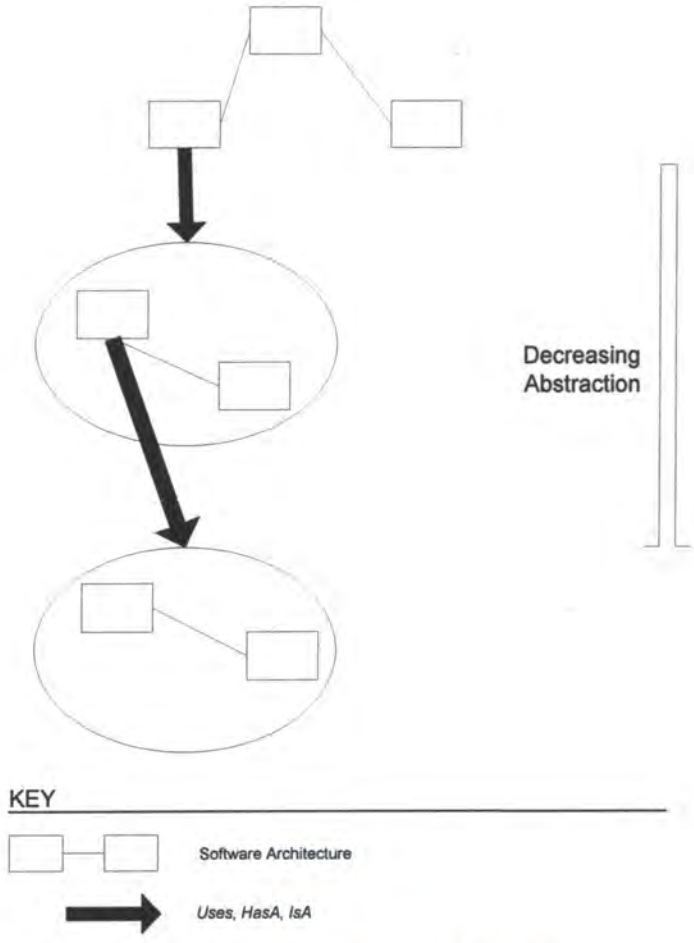


Figure 5 - Abstraction and Software Architecture

Software evolution at the level of abstraction provided by software architecture is concerned more with the configuration and integration of software components than with their construction. Software evolution is viewed as structural changes. A particular software architecture (or set of software architectures) forms the basis of any software system and typically stays with that software system throughout its lifetime. This has an effect on the evolution process, since it is the latter that has to be accomplished in light of the particular architectural style (or styles) chosen, even if the architecture no longer satisfies the requirements made of it. Architectural heterogeneity arises when software systems consist of different software architectural styles at different levels of abstraction [Garlan93a]. In this case software evolution may occur differently in different parts of or at different levels of abstraction within a software system, resulting in the need for different techniques in different parts of the system. For example, a two-layered software system consisting of an event-based architecture in the top level and an object-oriented architecture in the lower level would require two different techniques if evolution is to be performed. Additionally, the interactions between layers have to be addressed in the evolution process.

Cazzola et al have investigated evolution of software architectures, using as a case study the domain of a distributed traffic control system for a railway [Cazzola97a]. They use meta-level modelling (or a reflection approach) to structure potential evolution changes to the software by defining levels of evolution. A change that is not performable within the "adaptation space" defined by one level of the reflective tower is moved to the next meta level. The adaptation space for a particular level is defined by the set of components, connectors, operations and constraints defined within that level.

So, in the railway traffic control system example, the components are the railway tracks, the connectors connect the railway tracks, the constraints ensure that each railway track is connected at either end by only one other railway track and the operations allow one to add or remove a track. The adaptation space in this base layer prevents certain topologies such as a star topology. In order to allow this, the constraints must be relaxed by moving to the next (meta) level. The main problem with their work is the fact that the reflection and modelling approach used permits an adaptation space in which the only parameter (changeable aspect) is the topology, which is fairly inflexible in terms of the kind of evolution supported.

Choosing an architecture for a software system results in the chosen architecture being a design decision or assumption, which may conflict with future requirements, although domains can have standard architectures (as remarked above) and the architecture of a software system is unlikely to change drastically over its lifetime.

5.7.1 Object-Oriented Software Architectures

The imposition of structure by the modelling process can hinder the evolution of the software. For example, classes only allow methods in a particular class to call other methods that:

- Are members of the class;
- Members of a superclass;
- Members of any classes that are referenced by the class, and their super-classes.

Hence, there is an implicit dependency between the structure of the class model and the visibility of methods. If evolution requires a change in the visibility of methods, the class structure must be changed.

Class models, if viewed as a graph, also tend to have a fairly low connectedness. This means that extra references have to be put in if methods require it, causing further problems for evolution.

Having classes provides some semantics by allowing the software engineer to describe valid combinations of methods through the use of the visibility imposed by the rules described above. However, this can cause problems especially in the early days of software development when the requirements and therefore the design are constantly changing, because inaccessible methods need to be made accessible. This results in changes to the class structure which may percolate.

Method visibility in object-oriented models is determined according to the following rules:

- A method in class A can call any method in any superclass of class A;
- A method in class A can call any method in an object referenced from class A;
- A method in class A can call any method in any superclass of an object referenced from class A.

So, in Figure 6, method $A.m_1(\ldots)$ would be able to access the following methods:

- A.m* (...);
- B.m* (...);
- Super-A.m* (...);
- Super-B.m* (...).

These rules constitute an implicit relationship, "can-call", that determines which methods can be called from each method. Any changes to this relationship require changes to the object-oriented class model.

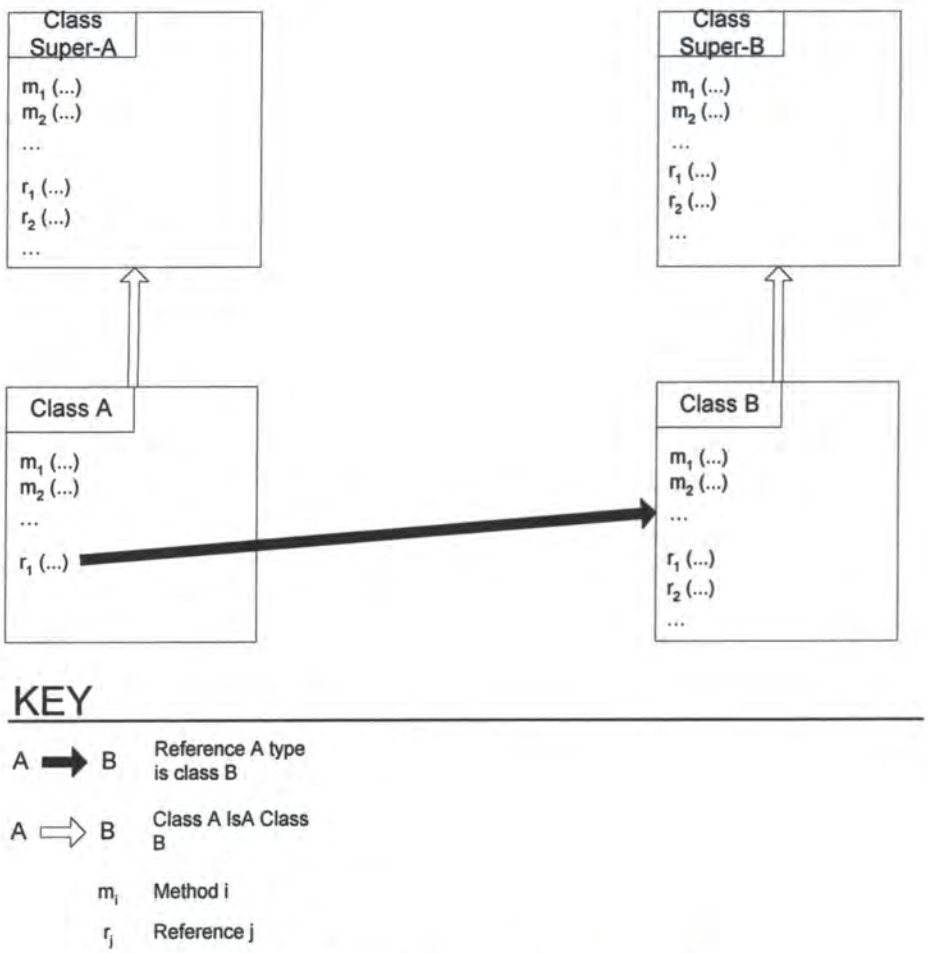


Figure 6 – Method Visibility in an Object-oriented Model

5.8 Software Development Techniques

Intentionally or not, current software architectures tend to be more geared towards software construction/development than software evolution and maintenance. As a result, the latter are difficult to perform. This is for various reasons. For example, object-oriented practitioners often extol the advantages of inheritance as a structuring mechanism in software development, without thought for the difficulties in understanding software designed and implemented in this manner, especially when many levels of inheritance are involved. In addition, the dynamics of evolution of object-oriented models are not very well understood. There is little theory on the effects of change in an object on those other objects which depend on it through inheritance and reference relationships.

The general approach to software evolution and maintenance is system comprehension (understanding the software) followed by making the change, typically in a very ad-hoc manner. Current software architectures and tools provide very little help to the software engineer making changes to the code. The process of evolution is ad hoc with no standard methods of performing it. It is also very general, providing no specific help or guidelines to help change the software. However, process is very dependent on a good product model in order to be successful. Traditional change-request-oriented maintenance and evolution process models rely on current software product models, which provide very little help to the process (and hence the software maintainer) for performing evolution, because the models don't encapsulate theory on how to perform evolution. Examples of some of the shortcomings of current software models are:

- The difficulty in both determining in-built assumptions and design decisions and altering them in the light of new requirements;
- The lack of both modularity and an understanding of the evolution relationships between abstractions in the software models i.e. what are the effects of a change in an abstraction on other abstractions in the software?

5.9 Lack of Modularity

Modularity is a form of software model structuring that provides a way of extracting common patterns (or abstractions), such as functions, data structures and architecture, and determining their dependencies. Current software modelling techniques and architectures, however, lack modularity because the abstractions they use are often mixed together and lack separation. For example, the notion of application domain is not modularised, so that domains are often implicit in traditional software and difficult to extract out. Similarly, control and functional capability are intermixed although, in this case, the inherent dependency between the two types of abstraction is stronger.

The mixing together of control and functional capabilities in software systems results in code that is difficult to understand and in which it is difficult to localise changes. For example, in a 2-D graph layout program, the functional capabilities:

- Graph layout and
- Graph display

and the code that controls the behaviour of the program are all intermixed into the same monolithic program. Even in an object-oriented software system, where there is some modularity in the sense that different capabilities should be separated, maintenance can be difficult because of the inherent complexity of the class structure.

The lack of semantic richness of current software models means that the set of construct types (or software entities) with which software is developed is quite low level. This lack of construct types has two main consequences:

- It is more difficult to map from the problem domain (the real world problem that is to be solved by the software) to the solution domain (the set of software entities which are used to construct the software) because there are fewer constructs to use;

- Changes are not localised. A classic example of this involves cross-cutting concerns [Ossher98a], which "...affect the definition of collections of operations that span multiple units of functionality." Ossher et al give the example of a print capability, which is a feature of many components of a computer system. For example, an email message component can be printed and a graphic component can be printed. However, the similarities between different implementations of a print operation cannot be gleaned from code because there is no way to determine their similarity. Hence, changes that affect all print operations are difficult and costly to make. For example, if printing should require an email to be sent to the system administrator, all print operation will have to be altered. A solution is to increase the semantic richness of code by including more information (possibly using types) for software entities. In this example, all print operations could inherit from a "print type" that would be the only software entity that had to be changed to satisfy the new requirement.

5.9.1 Software Entity Interfaces

A software entity typically has an interface which it exports to other software entities. This interface provides a well-defined interface to other software entities in the system, ensuring that the software entity is used in the right way and not accessed improperly. Typically, interfaces are functional, and in this case interfaces consist of message types that the function can interpret/parse. Message types include both data-oriented and control-oriented messages. However, there also exist more declarative interfaces which commonly provide a language or model for specifying what is to be done. The language or model then forms the interface. An example of such an interface is a data transformation, for which the interface provides a set of constructs for specifying how to convert data from one model or schema to another.

Software entities are generally written to perform correctly given a particular environmental context or set of environmental contexts. When this environmental context changes, software entities can not be guaranteed to work correctly. For example, a sort entity will export an interface consisting of the type of data that it expects. If the entity receives data that is not in the correct format, signalling a change in its environment, then it can not be expected to work properly, and evolution must be performed or an error signalled.

Interfaces mean that software works if the software entities adhere to the interface. But, this is an invalid assumption to make if evolution or an error occurs. In these cases, procedure calls will lie outside the interface hard-coded in the software and a new interface will have to be written.

5.10 Incompatibility Between Required and Existing Abstractions

Choice of abstractions conflict with changes, so that the interfaces provided by the existing abstractions are not enough. The abstractions are suitable at the time of design of the original software system, but changes in requirements mean that the abstractions need to be changed, which in turn may require new parameters which are outside the capabilities of the interface provided for the abstraction.

5.11 Coupling, Dependencies, Assumptions and Ripple Effects

Coupling is an inevitable characteristic of software, resulting from the fact that no software construct is an island and must depend on other software constructs in order to perform the task required by the requirements. However, software constructs contain implicit assumptions about the software constructs on which they depend, which causes problems when those software constructs change. For example, in object-oriented software, methods assume a particular structure for the class of which they are members, and assume parameters of a particular type and structure. Ripple effects occur when these assumptions are invalidated by a change in a part of the software on which another part of the software depends on or is coupled with.

Current approaches to ripple effect analysis (of which there is a good discussion in [Turver93a]) have been modelling the probability of a ripple effect, and determining if a ripple effect will occur. Little has been done on *what* the effects of ripple effects are, on the essential task of determining the *types* of ripple effect that can occur. Research into ripple effects is in direct opposition to work on adaptable and flexible software, which attempts to limit ripple effects by increasing the adaptability of parts of software with respect to entities (including software entities, users, hardware) in their environment by limiting the assumptions made.

5.12 Lack of Evolveability

The lack of evolveability of present-day software is caused by a number of factors, such as:

- Design decisions;
- Assumptions;
- Characteristics of the software architecture. For example, in an object-oriented model, classes constrain method visibility creating inflexibility in which methods a particular method can send messages to.

Flexibility is defined as the ease with which a software entity can be changed to satisfy new requirements. The difference between adaptability and flexibility is that adaptability deals with the extent to which **clients** of a software entity are able to adapt their assumptions about the software entity, whereas flexibility deals with the ease to which software entities can be changed. An example of inflexibility in software is that of object-oriented models, which constrain the visibility of methods. A method which is a member of class, C, is only able to call other methods that satisfy the following criteria:

- Member of C;
- Member of a superclass of C;
- Member of a referent of C;
- Member of a superclass of a referent of C.

A change in requirements that results in a need to call a method not satisfying this criteria means restructuring the model. Of course, there will be many potential ways of modelling a problem. As a simple example, a list can be represented or modelled as a linked list or an array. A particular object-oriented model may be more flexible with respect to methods

and a new requirement than another object-oriented model that implements the same problem. Hence, flexibility is dependent on the particular model chosen *and* the new requirement. There are different types of flexibility, including:

- Flexibility of method visibility: the ease with which method call targets can be changed;
- Flexibility of data structures: the ease with which data structures can be changed.

As Hillis notes in [Hillis98a p143], inflexibility is a characteristic of engineered systems like software, which depend on modularity in order to overcome complexity. It is a characteristic which can't be totally overcome because of the nature of engineered software. This means that the strict hierarchy and contract-based nature of software imposed by modularity makes it difficult to change the software so that a different hierarchy is produced.

Object-oriented software models allow software to be designed that is naturally cohesive, with low coupling. The constructs in the model allow these characteristics to be built into software, but depend on the software engineer's skill in designing the software so that these characteristics are maximised. Of course, success in achieving these characteristics improves the adaptability of the software, because:

- Increased cohesion focuses changes to as few objects as possible i.e. localises evolution. This is the same argument as for increased modularity through the identification of a richer set of software entities which have natural cohesion;
- Decreased coupling limits the effects of changes by limiting the dependencies in software.

Hence, the advantages offered by object-oriented systems when designed properly lies in the advantages of cohesion and coupling that they support.

However, the argument of this thesis is that, by an appropriately different type of model, such contracts and hierarchies can be relaxed so that a change in requirements can be effected by a smaller set of changes that don't break as many assumptions and hence produce fewer ripple effects. For example, rather than having service x directly call service y to perform a particular task, indirection can be utilised to broker the service call and provide a centralised interface (through the broker) to changing the mapping from required task to service implementation.

5.13 Lack of a Change Type Taxonomy

In practice, there are different types of changes. Often, changes are quite modest, though the effort of finding them, and determining what does changes and what does not change may be huge. Other changes may be inherently large, such as changes that result from many conflicts between the existing code and new requirements, or structural changes. All changes are typically ad-hoc in the way they are made. Certainly, software maintenance and evolution process models provide little in the way of detailed change management procedures because of their abstractness. There is a lack of theory on types of changes, stemming mostly from a lack of modularity and range of software entities/constructs (which provide targets for evolution and hence generate change types). There is also a lack of theory on how changes in parts of software affect other parts of the software which depend on them.

The set of software entities required to fulfil the new requirement is dependent upon the existing software entities of the system in the sense outlined above and expressed by the following:

$$\{\text{Software Entities}\}_{\text{New Requirement}} = \{\text{Software Entities}\}_{\text{New Requirement + Existing Requirements}} - \{\text{Software Entities}\}_{\text{Existing Requirements}}$$

Figure 7 - Relationship between Requirements and Software Entities[9]

This is why software evolution and maintenance are difficult, because the software engineer has first to determine which software entities exist in order to determine which further software entities are required.

- Determination of missing software entities;
- Integration of sub-systems, used for integration-type changes that require new software entities in order to satisfy a new requirement.

At present, the effort of making a change is dependent on the size of the system, not the change, because dependencies between elements of the software result in ripple effects. The existence of assumptions present in software as a result of the design process produces ripple effects when changes are made that invalidate these assumptions. There is no way of getting around this, even if the adaptability of particular parts of the software can be improved by appropriate modelling. For example, adding an "onhold" feature to a telephone system is easier when there is one switch than when there is more than one switch because in the latter case routing comes into play. In other words, introduction of a new service involves checking whether the new service is consistent with existing services, the so-called feature interaction problem typically found in telecommunications literature [Zibman95a]. Of course, this isn't limited to software and software engineering. Other branches of engineering suffer from similar problems. For example, improvements made to aero engines can often have a weight penalty.

This can explain why the effort of the change is greater when the existing system is more complicated, because the new change can potentially interact with any part of the existing system. Each new capability/service has to be checked for consistency with existing capabilities and the effort inherent in this increases in proportion to the number of existing capabilities. The effort of making a change is also dependent on the type of change. For example, the effort of making relatively simple parameter value changes is typically proportional only to the size of the change. However, even a parameter value change can result in wide-spread changes. Consider a function that writes out data to a disk and is of the form:

```
Function WriteDataToDisk (File F, GenericPointer Data)
```

[9] The software entities required as part of evolution ($\{\text{Software Entities}\}_{\text{New Requirement}}$) are those that remain after taking the set before evolution ($\{\text{Software Entities}\}_{\text{Existing Requirements}}$) away from the set after evolution ($\{\text{Software Entities}\}_{\text{New Requirements + Existing Requirements}}$). This is dependant on the *relationships* between the set of software entities after evolution and the set of software entities before evolution i.e. conflicts, specialisation, extension, removal etc.

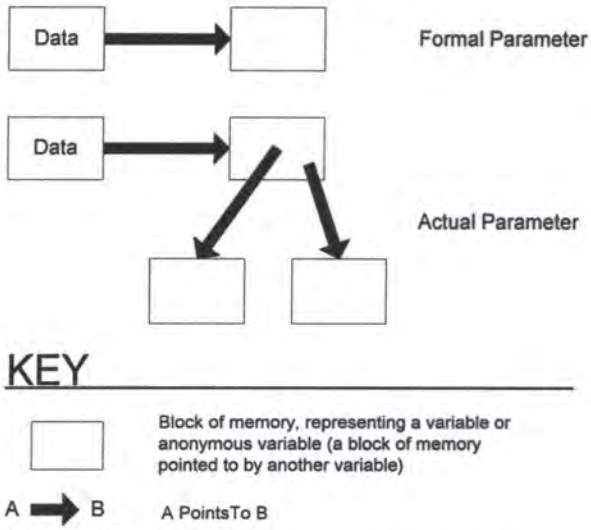


Figure 8 - Actual Parameters, Formal Parameters and Software Evolution

Note that the function assumes very little about the form of the data to be written to disk, other than that it is a block of data ("GenericPointer" is similar to, for example, the "void *" pointer type in C). Now consider an actual parameter in the form of a linked list, as shown in Figure 8. Since "WriteDataToDisk" assumes that "Data" points to a block of data, it will deal with the actual parameter incorrectly. This is a problem with the typing mechanism, in which "Data" is typed too abstractly (probably because the requirements on this parameter have changed from writing simple data to more complex data), but shows how a seemingly simple change can affect the behaviour of a program quite drastically.

The main problem is that the effort of making a change is dependent on both the existing software and the change itself, because the change depends on other capabilities which may or may not already exist in the software. If the capabilities do already exist then the effort of making the change will be easier, if not then the effort of making the change increases. Of course, a lot of time can be spent determining if the capabilities already exist.

There is also the problem of not knowing the extent of the effort that a change in requirements will bring about. Making a change may result in ripple effects that spread throughout the software, thus increasing the effort. A seemingly small change to the value of a parameter may, for example, lead to other changes having to be made. This depends on the type of parameter and may also depend on non-functional characteristics of the software. For example, changing the types of the data items to be sorted is a fairly simple change because it doesn't require changes to any other data or control part of the system. However, introducing a non-functional requirement such as a time constraint may result in further evolution (perhaps use of another more efficient sort algorithm) because the new type takes longer to sort/compare. Some of the data in a software system may "tweak" the control aspect of an algorithm so that a change in the value of this data will result in different behaviour. If the new value is not supported, further evolution will be necessary.

6 Summary, Discussion and Conclusion

In summary, the problems posed by current software evolution techniques are:

- Mapping the information contained in a change request to actual changes to the code (ignoring ripple effects, which are considered a separate problem). A major problem with current software development and techniques is that the concepts in the application (or real world) domain are not easily mapped onto concepts in the implementation domain. This is compounded by the fact that software engineers need to be able to understand the application domain in order to map the requirements in this domain onto the implementation domain (with which they are, of course, familiar). Automatic programming approaches attempt to address this problem of gap between requirements and code by utilising a very high level language (typically mathematically- or set-based, much like specification languages like Z [Potter91a] and VDM [Jones86a], but executable), but this still has the problem of requiring the software engineer to understand the application domain. In addition, these formal approaches fail to overcome the problems posed by Lehman's Laws. Domain specific languages seem a more promising approach and allow the expert in the domain (the user) to specify their requirements in the domain specific language, which is then mapped into an implementation by utilising a set of increasingly lower level languages arranged in a hierarchy below the application domain specific language [Ward94a];
- The difficulty in determining, given a new requirement, what does change and what doesn't change in the software. This is dependent on the existing software because:

  1. The new requirement may conflict with existing aspects of the software, which then need to be changed;
  2. There may be aspects of the software which already satisfy the new requirement.

  System comprehension is required in order to determine the answers to 1 and 2. In order for the software to be able to help in determining these answers, the requirement must be expressible in terms of a specification language which the software can interpret. Domain specific languages are closer to a specification language than existing languages. However, problems still exist. Firstly, the inability of expressing all future requirements in terms of them, resulting in having to resort to existing techniques. Secondly, extra effort is required in order to produce languages which effectively have redundancy built-in i.e. they provide more capabilities than required for a particular software project;
- Ripple effects, and the ability to determine how changes to the software affect other parts of the software, a consequence of the fact that changes to an aspect of the software may conflict with assumptions that other aspects of the software make about the changed aspect. For example, assumptions about the class structure;
- Lack of links between abstract concepts (in documents) and concrete concepts (code elements);
- Lack of evolveability (flexibility and adaptability) of existing software languages, models and architectures.

Chapter 3

Candidate Approaches to Easing Software Evolution

1 Introduction

Most improvements to the modelling of software have aimed at improvements in software development (for example, object-orientation provides encapsulation and inheritance to aid software development). In comparison, improvements haven't been adopted to aid software evolution. The measures built into software to aid software development don't, in general, help software evolution which requires different techniques. This thesis makes the hypothesis that there is no current model or architecture that inherently aids software evolution. Current software architectures result in software that is difficult to evolve, because:

- Software can't be understood simply by looking at it. Approaches to improve this include system comprehension techniques, the linking of documentation to the software elements that implement document elements [Karakostas90a] and domain specific languages;
- Requirements can't be directly expressed in most existing software languages. Domain specific languages increase the level of abstraction and allow requirements to be more directly expressed in terms of concepts at the level of the requirements, but are rigid when it comes to expressing requirements outside the constraints of the domain;
- Changes result in unwanted side-effects in other parts of the software system, because changes to a software entity alter the assumptions that dependent software entities have. A classic object-oriented example involves the assumptions that the functional aspects of code (the methods) make about the class structure – when the class structure changes, particular methods may be invalidated [Lopes94a];
- Inflexibility. The coding of requirements into code produces a form of inertia which makes it difficult to change software to satisfy new requirements that conflict with these hard-coded requirements;
- Lack of adaptability;
- Lack of *support* for extensibility;
- Lack of encapsulation of aspects of the code prone to change. Admittedly, such aspects are seldom known at the time software is developed, and only later do the changeable aspects of the code become apparent. Often, these aspects have not been encapsulated appropriately in order to utilise an interface to help overcome the effects of ripple effects.

There is a need to build software with evolution in mind from the start if the evolution task is to be eased, and overcome the problems introduced in chapter 2, namely:

- The inability to understand what software does and how it does it, except through a costly system comprehension process;
- Lack of linkage between the various phases of software development;
- Mapping a change request to a set of changes to the code;
- Ripple effects.

The following sections describe a number of approaches to software design, and software models that attempt to ease software evolution. They are all based on a number of common themes, namely:

- Separation of concerns, in order to improve adaptability and localisation of evolution;
- Transformational approaches, in which a high level requirements-oriented model is linked to a lower level implementation model (possibly through a sequence of gradually less abstract models) through a set of transformations or relationships.

which they all approach in different ways.

2 Separation of Concerns and Integration

Separation of concerns identifies a set of concerns (or modularisations or abstractions)[1] that together provide a set of constructs that can be used by software engineers when developing software. These concerns include but are not limited to:

- Components, including functions and their algorithms;
- Location of components;
- Asynchronous/synchronous nature of the software architecture;
- Concurrency;
- Exception-handling.

The main point is that these concerns are sufficiently context independent and can be modelled in isolation, or at least be modelled with some detachment from other concerns.

Concerns may or may not be expressible in terms of current software constructs, such as functions and components. Many of them will be expressible, but the mapping will be complex because they cut across many different constructs. For example, concurrency is not expressible in terms of any one component because it affects many components. Kiczales et als work at Xerox Parc has had some success in separating out such abstract concerns by providing separate notations for them [Hursch95b]. Hence, concurrency can be expressed in a separate notation than other concerns, such as the algorithms. The problem is that some concerns, such as concurrency, are inherently highly coupled to constructs in

[1] The entities produced using a separation of concerns approach will, from now on, be termed "software entities". The software entities used in SEvEn are described in chapter 5.

other concerns. This characteristic of some concerns stems from the fact that they encapsulate properties of sub-systems (consisting of many constructs) as a whole.

Hursch et al argue that development of software within one concern should not have to include thinking about other concerns as well [Hursch95b p5]. These other concerns should be abstracted out and worked on at a later time. This is the main idea behind separation of concerns, in which decisions about particular concerns or aspects of the code are delayed until work has finished on the current concern. For example, when writing an algorithm, the software engineer should not have to be concerned with the other aspects identified above. These other aspects should be abstracted out whilst the algorithm is being formed. The main flaw with this argument is that different concerns will often have high dependencies on each other i.e. their context dependence is high because they are specialised for a particular task (for example, a data structure may have a high context dependence on the existing software system with which it is to be integrated because it can only be used for the particular task for which it has been designed). In this case, the cohesion of the aspect is low, the context dependence is high and it is very difficult to separate out the dependencies.

However, the real power behind separation of concerns is the increased use of abstraction, in implementation terms hiding design decisions behind an interface which allows the "used" concern to be changed without affecting the "using" concern (unless, of course, the interface is affected by the change). As an example, consider the "location of components" concern identified above. Traditionally, the location of components such as functions is determined at link-time, so that functions have an entry point in local memory. This is handled by the compiler, which assumes functions are local. Separation of concerns would abstract out the decision of locality for functions behind an interface, which would form a barrier across which changes to the location of the function wouldn't pass. The caller of the function would call this new interface, which doesn't change. Any changes to the locality of the called function would be hidden behind the interface.

If software architecture is viewed as a concern to be separated then, like other concerns, there must be some way to separate out the code for software architecture from code for other concerns, such as concurrency, location of components etc. Software architecture code is concerned with a number of aspects of code that mainly have to do with the types of components and connectors used. Interfaces can be introduced so encapsulating the concern and provide for late binding between concerns. In the case of software architecture, interfaces can be used to encapsulate decisions about connectors. In SEvEn, connectors are realised in terms of messages (see chapter 5). Lopes discusses the separation of concurrency and remote access strategy and how these two aspects are integrated with the main code that performs the application task [Lopes97a p51]. She develops a language for each aspect, consisting of constructs that can be found in existing 3GLs, such as synchronisation and mutual exclusion for the concurrency aspects, and parameter passing semantics (such as pass by value) for the remote access aspect. The concurrency language has read-access to the main code's data, and the remote access aspect has access to the main code's data and functions. Hence, integration is performed through "use of" functions and data.

With any set of software entities, there will be inter-dependencies between individual software entities. For example, an object-oriented model consists of, amongst other things, classes, class models, methods, class data members, inheritance

relationships, reference relationships etc. These software entities are inter-dependent. For example, methods are dependent on class structure.

The increased use of different software entities means that individual software entities can be developed independently from other software entities and, through the use of appropriate interfaces and abstractions, bound with those other software entities as required. Viewed another way, such late binding involves deferring the decision about the implementation of a particular software entity until compile-time or run-time (rather than during software development). For example, when developing a function, rather than having to choose the particular connector semantics for those functions called by the function, late binding can be employed so that the choice of connector semantics is delayed until a later time. An explicit compile-time dependency is transformed into a looser dependency that isn't determined until later on, and is hidden behind a well-defined interface. Other examples include:

- Late binding of function to class structure in object-oriented software systems [Lopes94a]. The function is implemented in a class structure-shy manner, making as few assumptions possible about it so that the class structure can be later bound without major changes to the function's use of the class model;
- Late-binding of connector type (in software architectural terms, a connector type designates the type of message-passing semantics between procedures or functions or methods) [Bass98a p272]. The choice of connector type can be made to depend on desirable characteristics of the environment or quality of service (QOS) parameters, so that a change to the connector type can effectively be made by changing a parameter – a re-configuration type change which is arguably easier than an integration-type change;
- Late binding of software architecture, in which the software architectural elements of the code are separated from the application domain aspects. The application domain aspects can then be written in terms of a software-architecture-independent interface, which can then be bound to a particular architecture. For example, a compiler is expressed in terms of a lexical analyser, syntax analyser, code generator, optimiser etc. This example is a bit more complicated because the application domain code makes certain assumptions regarding software architecture.

Traditionally, such bindings are explicit. For example, function calls assume local, synchronous semantics. Changing this requires changes to the code at the point of the function call. The use of an interface behind which such changes occur improves the readability and modularity of the code.

The implementation of a software entity can be changed by changing the binding. The problem is that the alternative bindings must:

- Be compatible with the interface of the software entity being bound, and;
- Conform to the assumptions that both elements of the binding make of each other.

Separation of concerns results in a set of concerns that must be integrated together in order to provide a working software system. Lopes et al call this aspect weaving [Mendhekar97a p30]. It involves the use of integration knowledge in order to determine how to integrate the various concerns.

3 Transformation of Models

Hursch views software as consisting of three inter-dependent parts:

- Schema/conceptual model, corresponding to the class graph in an object-oriented model;
- Behaviour model, corresponding to the functional aspects of software, the methods and **propagation patterns**[2];
- Structural/Object model, corresponding to the object graph in an object-oriented model,

as shown in Figure 1 [Hursch95a p24]. Other similar classifications are possible, but all involve a similar classification into data and function. However, the classification should also include a structure- or software architecture-based aspect, which Hursch's doesn't because it is limited to object-based models in which the objects carry the burden of modelling the architectural aspects of the software. In Hursch's classification, the conceptual model provides the basis for change in the other two models (changes in the class structure affect the object structure and functions that use the classes). In addition, changes to the structural model can affect the behaviour model. Hursch addresses these problems by developing a set of rules that define how one model evolves with respect to another model. His results are limited to how changes to the conceptual model can affect the other models. Changes to the conceptual model are realised in terms of changes to a class model through the cross product of:

- Software constructs:
  - Classes;
  - Method signatures;
  - Class attributes;
  - Inheritance relationships;
  - Access control;
  - Composite (part-of) relationships;
- Change operators:
  - Add;
  - Remove;
  - Change/Rename [Hursch95a p54].

[2] A propagation pattern is, crudely put, behaviour that extends across many classes.

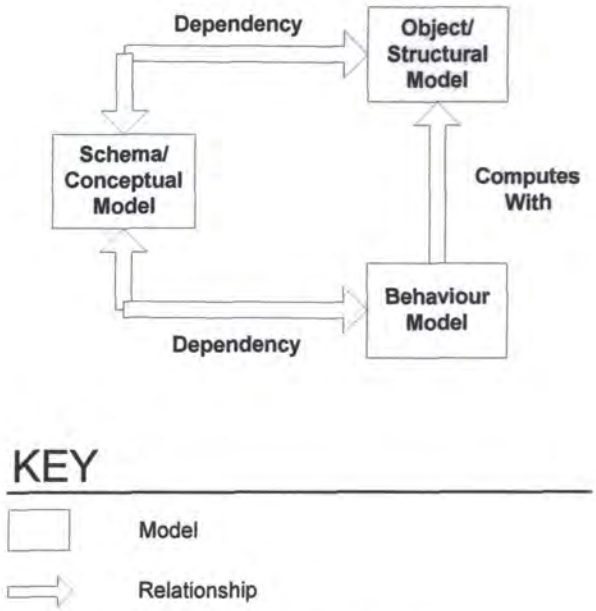


Figure 1 - Hursch's Modularisation

Hursch's work can be generalised to that shown in Figure 2. The abstract system model represents the software system at a reasonable level of abstraction. In this context, the term "model" is a generic term that can mean a language, a graph, or some other parse-able model. With appropriate substitutions for the abstract system model and the concrete sub-models, the model as a whole can be made to represent:

- Traditional approaches, where the system model is a programming language, and the concrete sub-models represent the hardware constructs;
- Domain specific languages (see section 4), where the system model is the domain specific language, and the concrete sub-models represent the underlying components that implement the domain language constructs.

The system model is more abstract in the second case than in the first case.

The change is made at the system model level by the software engineer. The system model is hopefully at an appropriate level of abstraction so that requirements conflicts can be determined at this level. The dependencies of the lower level models on the system model ensure that changes made at the system model level result in appropriate changes in the lower level models. In essence, the architecture provides a way of linking high level concepts in the system model which are close to the requirements, with implementation-level concepts. With an appropriate system model, requirements can be expressed readily and conflicts determined.

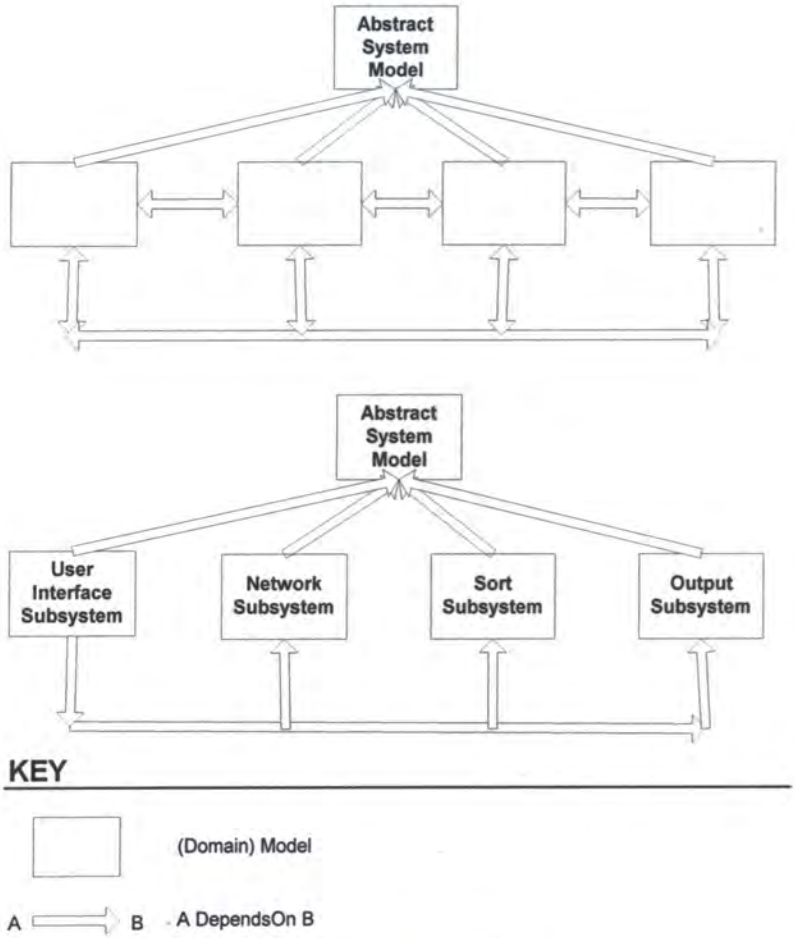


Figure 2 - Evolution of Models

These approaches depend on the software engineer being able to express the new requirement in terms of the system model. In other words, they concentrate on *how* software evolution occurs rather than how the new requirement is specified. The system model may encapsulate what the model is capable of, but such knowledge is typically implicit and tied up with the orthogonality of the language. For example, the scope of what programming languages such as C are capable of doing is determined by:

- The constructs of the language;
- The orthogonality of the language i.e. the degree to which the constructs can be combined.

Most programming languages aren't very orthogonal and it isn't always clear how their constructs can be combined to produce higher level constructs. This means that it may be difficult to determine what the language is capable of doing (beyond what it is already being used to do). An approach based on the explicit modelling of what the system model is capable of doing, its extension, may help in this regard.

Much of the work on changes in models derives from research in changing database schemas, specifically in object-oriented database systems [Clamen94a], but also in relational databases and the relational model for which the motivation for normalisation came from a desire to prevent ripple effects when the structure of a table is changed. Tresch and Scholl classify database schema changes as follows:

- Capacity preserving changes, which neither reduces nor extends the modelling power of a schema;
- Capacity reducing changes, which reduce the modelling power of a schema;
- Capacity augmenting changes, which extend or enhance the modelling power of a schema [Tresch93a].

Hursch presents a similar classification in [Hursch95a]. Both classifications describe change classes which have the characteristics shown in Table 1 (in which "capacity" means data modelling capacity, a characteristic of data which refers to the limits or boundaries of what can be modelled using the data model).

Change Class	Characteristics
Capacity Preserving	Dependent clients shouldn't be affected by the change
Capacity Reducing	Dependent clients may be affected by the change
Capacity Augmenting (Increased capacity)	Dependent clients shouldn't be affected by the change

Table 1 - Change Class Characteristics

A finer change type classification can be arrived at, which depends on the model being used. Schiefer describes a change type classification similar to that of Hurschs [Hursch95a], both of which use an object-based model [Schiefer93a]. An amalgamation of their classifications is shown in Table 2.

Type of Change	Comments	Class of Change
Add abstract class		Capacity augmenting
Remove Abstract class		Capacity reducing
Move reference to subclasses		Capacity preserving
Move common reference from subclasses to superclass		Capacity preserving
Add concrete class		Capacity augmenting
Remove concrete class		Capacity reducing
Add reference		Capacity augmenting
Remove reference		Capacity reducing
Specialisation of reference	The class in a reference is specialised	Capacity reducing
Generalisation of reference	The class in a reference is generalised	Instance-specific (a generalisation means that modelling power provided by the original class of the reference may be lost, but open up new modelling power provided by other subclasses of the reference's new class)
Rename reference		Capacity preserving (any renaming doesn't alter semantics)
Add subclass		Capacity augmenting (because of new

		capabilities offered by the new subclass)
Rename class		Capacity preserving
Telescoping of inheritance	A new class is inserted between two existing classes	Capacity augmenting (because the new class may offer new capabilities)
Telescoping of reference	Class X referencing class Y is transformed into class X referencing class Y through class Z.	Capacity augmenting (the intermediate class, Z, may offer more capabilities than class Y)

Table 2 - Change Types

The use of different models with improved evolution characteristics and ultimately transformed into implementation-level models such as programming languages allows changes to be made in these different models and their advantages utilised. For example, object-oriented models typically constrain the visibility of methods by building in a particular structure that may be invalidated by evolution triggered by new requirements. An alternative model may employ very flexible relationships between classes so that method visibility is not restricted and every method can call every other method (with appropriate data conversions in place). This model could then be transformed into an object-oriented implementation model, or used as-is with an appropriate broker architecture in which the brokers provide the knowledge for linking method calls to methods. This approach provides the advantage of improved flexibility, at the cost of increased use of indirection and, therefore, probably decreased efficiency.

4 Automatic Programming and Domain Specific Languages

The primary aim of automatic programming is the specification of software behaviour declaratively as a goal which is then turned into a plan (sequence of actions) which satisfies that goal and produces the desired behaviour. This has been a primary aim of the AI community for years, but has been difficult to master in general. Taking a domain-oriented approach to language design essentially increases the level of abstraction of the software to a declarative level that the domain specific compiler then converts into machine code to be run on the computer. This then is essentially constrained automatic programming.

A domain can only be modelled or used if there is a common consensus about what makes up that domain. Most researchers define a domain as a set of concepts and their relationships about which a community or set of people agree upon. In small domains such as sort and 2D graphs, this may be a relatively simple task to do, since these domains are well defined and understood. However, domain boundaries start to blur when the domains being considered increase in size and complexity. For example, a typical telecommunications domain[3] is relatively large and continues to become larger in general, due to telecommunications companies coming up with new services. It is difficult to stabilise such a

[3] The phrase "a telecommunications domain" as opposed to "the telecommunications domain" is usual to indicate that there may be different domains coming under the general banner of telecommunications. Different people have different definitions of a domain, making domain boundaries somewhat blurred.

domain due to its inherent need to change in order to stay useful, as Lehman's Law of continuous change states [Lehman85a]. Taking an arbitrary point in an arbitrary domain's evolution (e.g. a model of a domain at a specific point in time) this domain can't be assumed to be stable. However, domains can be modularised into sub-domains so that the sub-domains themselves are stable. For example, POTS is fairly stable, even though the wider telecommunications domain of which it is a part will probably change. Tepfenhart observes that in the telecommunications domain, it is often difficult to determine if two software systems are part of the domain because these software systems utilise specialised sub-domains of the same domain [Tepfenhart97a p32].

As discussed, this problem can be overcome by splitting the domain into a domain hierarchy, producing smaller domains that can be made more stable and shifting the evolution that would occur within the domain to the creation of new domains that must then be integrated into the existing hierarchy. For example, adding a new telephone service "onhold" to an existing software system is traditionally approached by adding the new service into an existing domain. This integration process occurs through common concepts e.g. telephone numbers. There is, however, a problem with determining domain boundaries and what constitutes a domain. For example, is domain membership based on cohesion of concepts (so that a concept is a member of a domain if it uses more of the domain's concepts than any other domain) or some other characteristic?

There is a need to design domain-specific languages (DSLs) at the right level of abstraction. If too much is assumed in the language and those assumptions change then the DSL will have to change too i.e. its model will have to change. This implies that the same can be said of models i.e. models should be designed with economy in mind – don't model more than is needed in the core of the model (where the core of model refers to the part of model that has no interdependencies i.e. each part of the core is independent). For example, consider a telephone DSL written in terms of connect, disconnect and telephone number type. If a new service such as "onhold" is required which conflicts with assumptions made by the existing concepts then one may need to change the whole of the language/model. A classic example of this problem is in the domain of spreadsheets. Spreadsheets are a domain language for the world of rows and columns of figures. But then people start to use them for purposes for which they were never defined – such as accounting, really a separate domain which is incompatible with the world of rows and columns of the spreadsheet domain. At the other end of the spectrum, successful domain languages include CAD languages and languages that provide a way to perform electronic circuit analysis. These languages are successful because they encapsulate the concepts required to express a problem in the domain very well. More importantly, these domains are fairly static, so that the knowledge about the domain which they encapsulate is not likely to change and hence the language itself is static.

A major problem with domain languages is that any computer language is extremely difficult to design, and requires people of the highest calibre to do it properly.

An automatic programming approach can either be general and only partially automated, or model a narrow domain and be more automated. In the latter case, the automated programming system takes the form of an abstract machine (e.g. a domain-oriented language) in which the user specifies what he wants doing within the constraints imposed by the machine. The machine then translates this specification, using the assumptions built into the machine, into a working program in the low-level machine's native language. Problems occur, however, when the implicit assumptions built into

the abstract machine become false because of some change that is required by the user. The abstract machine essentially has a built-in model, which abstracts out some of the complexity of the domain, complexity which must be confronted when changes within the domain model need to be made.

Changes which can't be described in terms of any existing knowledge in the software system are difficult to make, because any "dialogue" between the software system and the software engineer trying to make the change will have no common basis of understanding. This is usually a sign that bottom-up evolution is required, because the basis for integration needs to occur at a lower level of abstraction, by the introduction of lower-level capabilities on which the desired changes can be built. Top-down evolution will only be successful if the required lower-level capabilities are present to support the new change. To overcome this, DSLs build in inherent redundancy so that they can be used to model many different types of real-world environment.

Ward suggests the use of domain-specific languages as the basis for a form of software development which he calls middle out development [Ward94a]. The basic idea is to develop a language at a suitable level of abstraction which uses domain concepts and which is more abstract than current languages. Then software development proceeds by developing the software in terms of the language. It is different from top-down development which proceeds from the abstract to the concrete, and bottom-up development which proceeds from the concrete to the abstract because development proceeds by developing both the language and the description of the required system in terms of the language. However, it is really only a layered approach to software development in which the domain language forms a layer upon which the software system is built. Plus, it has consequences for evolution when the underlying assumptions built into the language change. This must inevitably result in changes to the language itself, and we are back to square one. Of course, if applied to stable domains, then this approach will not result in changes to the language itself. One could solve this in unstable domains by constructing a number of languages at a lower level of abstraction, at which the constructs are stable. However, this assumes that the software engineer knows which constructs are stable and won't change. This approach is related to the discussion in section 3 on transformation of models where, in this case, languages take the place of the models.

Domain specific languages (indeed, any language for that matter) hard-code assumptions into the language which are hard to change after the language has been created/modelled. Of course, some assumptions have to be made, but making too many assumptions or the wrong assumptions will result in the domain specific language having to change. The trick lies in making assumptions that won't change. Of course, this is impossible in practice. Indeed, software engineers don't always know that they're making an assumption. Making assumptions is a fact of life for software engineering, even in traditional third-generation languages such as C, where functions assume particular types for their parameters and interfaces for the functions that they call. Every assumption can't be removed because of the nature of finite models – a finite model (that is, any artificial model) can't be used to model an infinite model (such as the real world) without making assumptions about the infinite model [Lehman99a].

Domain specific languages are closer to specification languages, in which a new requirement can be directly expressed in terms of elements of the domain language. However, domain languages can only be used to specify requirements that they can implement. So, they can't be used to determine whether a new requirement requires new components or not,

because there is a one-to-one mapping between the language and the underlying implementation of components. If new requirements aren't expressible in terms of the domain language, how are they mapped to a set of changes to the software?

Hence, in summary, domain-specific languages are not very good for dealing with:

- Unstable domains;
- Integration evolution,

primarily because the model is hard-coded into the language and difficult to change. In addition, today's domain-specific languages are typically not very evolveable.

5 Top-Down and Bottom-Up Evolution

Evolution spaces are fairly unrestrictive in that they allow either a top-down or bottom-up approach to software evolution. Using a top-down approach implies that the models used employ fairly abstract software entities at the top levels of the graph model so that top-down evolution is achieved by introducing new software entities below these levels through specialisation. It is important to choose an appropriate set of software entities so that future specialised software entities are compatible with the interface that they provide. This requires an analysis of the domain in order to determine the similarities between concepts so that they can be extracted out. For example, the telecommuncations domain basically consists of a telephone number data type along with a set of services such as "connect", "disconnect", "putOnHold" and "takeOffHold" that have certain functional similarities. These functional similarities include the following:

- An event that originates in a handset;
- Use of the telephone number data type;
- Similarities in trigger conditions. For example, whilst the trigger conditions on each service above are different, these trigger conditions do possess a common "parent" class. The trigger condition of "connect" is "dial tel-number" whilst the trigger condition for "disconnect" is "hangup", both of which are handset capabilities;
- Similarities in pre-conditions.

A model can be constructed based on these functional similairites, much like the functional similarity model used by Kishimoto et al [Kishimoto95a]. This approach is top-down because it allows evolution to proceed from an existing abstract model to an implementation that satisfies the new requirements. A top-down approach allows the software maintainers to begin from a fairly abstract starting point (that encapsulates common knowledge about the services in the domain) and then refine this to the required service. It does mean, however, that the evolution spaces are fairly large because the higher levels of the models contain abstract concepts. The success of a top-down approach is also dependent on the success of the modelling of the abstract software entities capturing the salient aspects of the domain. If these are incompatible with software entities which specialised them (in terms of interfaces, assumptions made etc.) then problems will ensue; in particular, the abstract software entities will have to change in order to accommodate the new

software entities, thereby invalidating existing software entities that specialise them. This is similar to the fragile base class problem in object-oriented models [Mikhajlov97a].

A bottom-up approach, on the other hand, uses more concrete models so that as a result the evolution spaces are smaller. However, their primary disadvantage is the lack of a good starting point for evolution. As a consequence, evolution tends to be more of an integration evolution approach.

Evolution spaces can be used in a number of areas in a software system, from architectural evolution [Cazzola97a], to control-based evolution. Indeed, it can be applied to any aspect of a software system that has the following characteristics:

- The aspect has an interface;
- The aspect has features which can be added, removed or changed, within well-defined (semantic) constraints.

The main advantage of evolution spaces is for system comprehension and determining if a new requirement can be satisfied by the current set of software entities that are available. Indeed, they are useful because they give the software engineer a head start on the program comprehension and evolution process by indicating how the *current* configuration can evolve. However, as stated above, they help only to evolve the current configuration. Any evolution that requires a more general interface will require more extensive changes.

6 Software Architectures and Models

There are product- and process-based approaches to easing software evolution. Process-based approaches are typically dependent on the underlying product and deal with the management of change. Product-based approaches, in comparison, deal with the underlying targets of evolution, and the design of the software itself. The design of software is heavily dependent on the software architecture or model chosen, of which there are many varieties [Garlan93a]. This section describes a number of product-based approaches to modelling, from complete architectural styles to specific modelling techniques, with the aim of describing their advantages with respect to easing software evolution.

6.1 Aspect-Oriented Programming (AOP) – Xerox Parc

Aspect-oriented programming is another approach to increased software modularity, which approaches the problem by identifying a set of concerns or aspects that are minimally-coupled [Hursch95a]. The aims are:

- To determine aspects which span multiple implementation-level concepts, such that changes to aspects affect many parts of the software system. The advantage of this is the evolution power that such changes possess;
- To be able to localise changes to the appropriate aspects, without having to be concerned with other aspects, by finding aspects which have little inter-dependent coupling.

In [Lopes97a p51], Lopes and Kiczales point out the difference between a software entity and an aspect:

> "...an issue that must be programmed is:
>
> - A component [or software entity in this thesis], if it can be cleanly encapsulated in a generalised procedure (i.e. object, method, procedure, API). By cleanly, we mean well-localised, and easily accessed and composed as necessary...
> - An aspect, if it can not be cleanly encapsulated in a general procedure. Aspects tend not to be units of the system's functional decomposition, but rather are properties that affect the performance or semantics of the components in systemic ways.".

It is clear that Kiczales et al don't yet have a clear definition of an aspect, nor a well-defined description of the differences between an aspect and a component. However, what is clear is that an aspect is not a component and a component is not an aspect. However, in the same paper, Lopes goes on to describe two aspects (concurrency and remote access) in terms of a component-based formalism, admittedly with different syntax and constructs than components in traditional software languages.

A major problem with Aspect-Oriented Programming is finding aspects for which the low coupling requirement is fulfilled. Of course, finding such aspects would result in gains in evolution localisation, but coupling is an integral part of engineering and abstraction, in which no abstraction is an island but is dependent on other abstractions.

6.2 Intentional Programming (IP) - Microsoft

The chief idea behind Intentional Programming [Simonyi96a] is that of separation of concerns. The argument is that the main computation should be separated out from other aspects of the code such as concurrency and remote access concerns, much as Aspect Oriented Programming and other approaches employ. As Simonyi states:

> "From the programmer's point of view, intentions are what would remain of a program once the accidental details, as well as the notational encoding (that is the syntax) had been factored out. Intentions express the programmer's contribution and nothing more." [Simonyi96a]

In essence, the problem domain notation (the requirements) is separated out from the solution domain notation (the implementation model or language) and bridged by procedural transformations called "xmethods". Hence, Intentional Programming is also related to transformational approaches to software evolution [Ward94a] because the core behaviour (expressed as a tree of DCLs[4]) is transformed into a tree of primitive executable constructs.

A related idea is that of the "personality pattern" [Blando98a], the intent of which is to increase the adaptability of behaviour with respect to class structure by attempting to extract out the skeleton of the algorithm, or the programmer's intent as Simonyi describes it.
Simonyis' main argument, however, is that one can't program purely in terms of problem domain abstractions; at some point lower level factors such as efficiency and compatibility with existing software systems and hardware will come

[4] A DCL is a "declaration of an intention".

into play. He provides a framework for expressing these different aspects (or domains) of a software system with as much separation as possible.

There is, however, little evidence of the scalability of the approach. Another problem is one of notation and determination of the domains that are important for use in modelling the intention. The software engineer needs to have a good understanding of which domains are at work in a modelling problem and which of these domains contributes to the intention, and which ones are accidental or ancillary domains. There also seems to be a lack of tool support for the non-trivial task of "xmethod" (the procedural transformations that convert the intentional notation into an implementation) development.

6.3 Subject-Oriented Programming (SOP) - IBM

Subject-Oriented Programming provides another approach to software modularity, in which the basic abstraction is the "subject" [Ossher94a]. A subject is a class model which represents a particular aspect of the problem domain in a subjective way. The framework provides:

- A set of composition operators which allow subjects to be composed in well-defined ways to produce a new subject;
- A method of mapping a subject to an implementation.

Hence, Subject-Oriented Programming is concerned with the composition of class models.

The main problem with this approach is that it's limited to subjects expressed using an object-oriented model. As the authors point out, however, the approach has potential for coping with evolution in which unplanned extensions can be accommodated by composing subject-expressed representations of the extensions with the existing subject-expressed software system [Ossher94a]. Another problem is that there are no clear rules for the software engineer about what makes a subject and currently no guidelines for how to develop subjects and recognise their characteristics. In addition, the approach seems to assume homogeneity of class models, in the sense that individual subjects must use fairly similar concepts for the composition to be successful.

Ossher et al use an example based on a trucking company which comprises two subjects:

- A "shipping" subject, which views trucks in terms of resources for carrying goods and is implemented in terms of a class which allows goods to be added to and removed from a truck;
- A "transportation" subject, which views trucks in terms of entities which can travel and is implemented in terms of a class which allows routing information to be set and queried for a truck.

The composition of these two subjects results in a subject which consists of trucks viewed as shipping *and* transportation entities. Notice, though, that composition assumes that the two subjects both model trucks using the same abstractions.

Also, it is unclear of the implications of name clashes in subject composition, an area which causes problems for multiple inheritance [Hansen90a].

6.4 Adaptive Software

Interfaces have been recognised as a well-proven technique for coping with change, such that changes are hidden behind interfaces. The problem is in designing interfaces that don't change. Another approach is to live with changing interfaces, but limit the effects of interface changes to specific parts of the software by introducing indirection into the software in the form of brokers, which cushion the effects of change.

Adaptability has been defined elsewhere (see chapter 2 table 1). Adaptive software deals with techniques of increasing the adaptability of parts of software with respect to other parts on which they depend. Most current approaches to creating adaptive software are either based on patterns or have resulted from specific patterns. These patterns are typically process-based, dealing with how to *write* software using existing architectures and models that improves the software's flexibility, an approach that depends on the software engineer's skills. This thesis, in contrast, aims at product-based adaptability, by identifying software entities with increased adaptability to change. The main pattern from which other adaptability patterns are derived is the "Inventor's Paradox" pattern, which, when solving a problem, proposes solving a more general problem. Applied to software engineering, the aim is to produce the most general software entity possible in a given context, for example the most general data structure or the most general behaviour. This is a technique well-known to the UNIX community in which a similar approach has been taken to the UNIX tool-based environment. The supposed advantage of this approach is that a more general problem will be simpler to solve. However, generalising a problem also typically removes assumptions and the interface of the software entity, so that more changes in it can be adapted to by any dependent software entities. For example, generalising a data structure lessens the assumptions that functions make of the data structure, so that particular changes in the data structure can be adapted to by the function. In particular, the function doesn't need to change in response to these particular changes in the data structure. A typical example of generalisation is the introduction of polymorphism into particular parts of the data structure. The generalisation process must, however, ensure that the resulting software entity conforms to the requirements that its dependants make of it. Hence, generalising a software entity too much may invalidate requirements that a particular dependant makes of it. As Lieberherr points out, a key to generalising a software entity is to avoid over-specification in software [Lieberherr94a].

The patterns that implement the "Inventor's Paradox" pattern include the following:

- **Structure-shy traversal** : de-couple behaviour from the underlying class model, in order to increase the adaptability of behaviour with respect to changes in the class model. As Lieberherr points out, however, this pattern depends on the software engineer identifying the static (unchanging) aspects of the software architecture encoded in the class model and designing the actual class model in such a way that these aspects are the only ones used directly in the behaviour of the software. The changing aspects are then used through an indirection interface (or broker) so that when they change, the ripple effects are limited to the broker. Identifying the static aspects may not be possible, or may fail to correctly predict the effects of evolution on these static parts [Lieberherr96a];

- **Structure-shy object** : de-couple objects (class instances) from classes, in order to increase the adaptability of objects with respect to changes in the classes on which they depend. This pattern is related to work in the database community on increasing the adaptability of the data in databases to changes in the schema (in this case, the data substitutes for the objects and the schema substitutes for the class model) [Lieberherr96a];
- **Context/Selective Visitor** : change the behaviour of function without changing the function, by using a context entity that describes when the change in behaviour should be triggered. The context could be an event. The change in behaviour, however, is restricted to behaviour extension i.e. changes that extend the behaviour without conflicting with the existing behaviour of the function [Seiter96a];
- **Class-graph**: automate the definition of class-graph induced operations such as copying, displaying, printing, checking etc in order to increase the adaptability of behaviour with respect to changes in class structure [Lieberherr96a].

The problem with pattern-based approaches to adaptability are that they are generally process-based and rely on subjective measures. For example, the personality pattern [Blando98a] and intentional programming (described above) are dependent on the software engineer to identify and separate out the skeleton behaviour of the code from other aspects, such as coupling with specific data structures.

Adaptive software entities adapt to changes in their environment (which consists of other software entities – see chapter 5 section 2.1.2). Much research has been carried out at NorthEastern University (the Demeter project) and Xerox Parc on the adaptability of particular types of software entity to changes in other software entities on which they depend, mostly concentrating on the adaptability of function with respect to data. But, there are other broad types of adaptability driven by the three main characteristics of software:

- Function;
- Data;
- Structure or software architecture, which stems from the overwhelming use of abstraction in software engineering.

Hence, function can adapt to changes in structure as well as changes in data, with appropriate modelling. There are, however, more subtle forms of dependency at work in software, as the remaining chapters explore. A prime example is the dependence of functions on specific functions which they use to perform a task. Changes in this mapping are difficult to make because of the inherently hard-coded mapping.

A major problem with software evolution is determining the effects of a change on other parts of the software system; specifically, determining what must change and what doesn't change. The evolution of a software entity can have ripple effects on other software entities in the software system which depend on the evolving software entity in some way. For example, the evolution of data structures can have ripple effects on the functional software entities that use them. The ripple effects must be dealt with in order to return the existing software entities to a state of consistency with respect to the original requirements. This is complicated by the fact that new requirements may conflict with requirements coded into the software system, making ripple effects inevitable. There are two approaches to dealing with ripple effects:

1. Built-in adaptability of software entities to changes in their dependants, typically by the de-coupling of the dependent and its dependants. For example, Demeter provides built-in adaptability for control constructs to changes in the class structure by the use of "structure-shy" control constructs called propagation patterns [Lopes94a];
2. "Post-adaptability", whereby specific transformations are applied to the dependent after one of its dependants has changed.

The choice of which one to use is dependent on how successful the extraction of assumptions has been. If it has not been successful, then approach (2) needs to be used.

Database research has produced many results in the area of adaptability [Clamen94a], mainly in the specific area of adaptability of instances to changes in the database schema. Some systems target changes in the methods that use the database [Banerjee87a]. Most approaches use transformations to provide a link between changes in the target and changes in the source that depends on the target (such as the example above, in which the instances are the source and the database schema is the target). The prototype implementations provide a set of standard transformations (such as changing the primitive type of a data element), whilst allowing the user to provide their own transformations based on particular changes to the target entity.

A major disadvantage of adaptive software is the inherent predictive element which requires the software engineer to determine the most generic interface to be used. This is predictive because the design of the generic interface must involve predicting how the abstraction with the interface may evolve, so that the interface may be designed in such a way that it doesn't need to change in response to changes in the abstraction.

6.5 Adaptive Programming (AP) – NorthEastern University

Adaptive Programming emphasises the adaptability of behaviour with respect to a particular context:

> "Instead of viewing a data structure as an integral part of an algorithm (as is done traditionally), we view it as context that can be changed significantly without modifying the program." [Lieberherr96b p77]

The types of context identified are:

- Data, class structure;
- Input to program;
- Run-time environment: "Depending on the other processes with which the program runs, the program will optimise a parameter to achieve better performance" [Lieberherr96b p77];
- Concurrency properties: separation of concurrency from the software algorithm;
- Software architecture;
- Exceptions;
- Computational resources e.g. printers, displays etc. not hard-wired in.

AP is related to separation of concerns/modularisation i.e. separating out dependent parts of the software. It is also related to late-binding (the behaviour is bound to a particular context at run-time) and to interfaces i.e. replacing a context with another one requires that the replacing context has the same interface as the replaced one - assumptions about interface. In addition, it is also related to parameterisation, that is shifting changes from changing code to changing parameters.

Additional types of context include:

- Type of message-passing style;
- Task knowledge (what to do) and function knowledge (how it is done)

> "The main goal of adaptive programming is to separate concerns by minimising dependencies between the complementary views, so that a large class of modifications in one view has a minimum impact on the other views." [Lieberherr96b p78]

In Demeter (the project at NorthEastern University that considers behavioural adaptability with respect to class structure), the contexts are expressed in terms of class graphs:

> "Adaptive software consists of three parts: succinct sub-graph specifications, C, initial behaviour specifications expressed in terms of C, and behaviour enhancements expressed in terms of C." [Lieberherr96b p80]

The key idea contributing to Adaptive Programming is the **Inventor's Paradox**: the approach is to write a more general program, which is similar to the UNIX philosophy and the idea behind open implementations and some aspects of language reflection [Kiczales91a]. The application of this pattern to Adaptive Programming is that in Adaptive Programming one is trying to find a generalisation of the data/class structures used. The supposed advantage is that it's easier to write for a more general data structure than a specialised one. The problems are:

- Finding the more general data structure;
- Mapping the generalised data structure to the more specialised data structure;
- Determining representation/interface independence. For example, functions/parameters and classes/implementations. In Adaptive Programming, this involves separating the behaviour from the details of data structures. However, behaviours are often intimately tied to their data structures, which limits the range of changes that can be made to the data structures without affecting the behaviour. This is also tied in with how constraining the requirements and assumptions which behaviour makes of the data structures; or highly constraining requirements and assumptions leave little room for evolution of the data structures without breaking the behaviour. As the Adaptive Programming principle states:

> "A program should be designed so that the interfaces of objects can be changed within certain constrains without affecting the program *at all*" [Lieberherr96b p81].

The main problem lies in identifying and enforcing these constraints.

Propagation patterns are a multi-class control traversal construct which allow the software engineer to express behaviour that inherently spans many classes. They're utilised in Demeter to improve the adaptability of such behaviour with respect to changes in the underlying class structure. Propagation patterns aren't totally adaptive, because they require changes to be made to them for particular types of change to the underlying class structure [Hursch95a p219]. Additionally, it isn't clear whether they're powerful enough to be able to capture the range of control constructs available in traditional programming languages. For example, how are loops and conditionals represented? Their supposed power lies in the fact that they protect functions from changes in the underlying class structure, but they rely on method calls being represented at the level of propagation patterns. It isn't clear whether this is always possible. Consider the class structure shown in Figure 3 and the method shown in Figure 4.

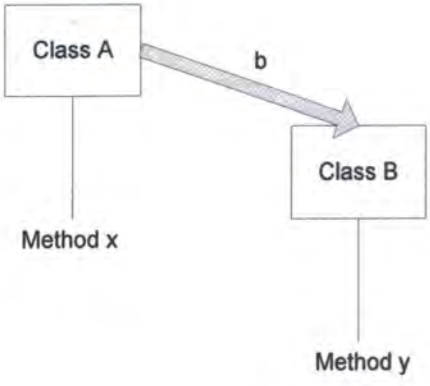


Figure 3 - Example Class Structure

```
void A::x (Parameters) {
        int i;
        ...          ———————————— A
        i = b.y (Parameters);
        ...          ———————————— B
}
```

Figure 4 - Example Method

In this case, method A::x consists of a call to method b::y. If the class structure changes, this can have an affect on the validity of the resulting method with respect to the original requirements[5]. This call isn't part of a propagation pattern and so isn't covered by the evolveability that propagation patterns offer. Of course, the code could be altered to that

[5] Which must still be satisfied in spite of the new requirement, excluding new requirements which conflict with existing requirements such as the introduction of call waiting in the presence of call redirection in a telephone system.

shown in Figure 5 and incorporated into a propagation pattern, ensuring that A::x1 and A::x2 are called in sequence, but this is messy.

Lieberherr et al build in adaptability by writing code in terms of the most general data structure possible, so that certain kinds of evolution won't break the code. The code is then bound to the particular data structure used, which is a specialisation of the data structure which the code is written in terms of. There are a number of problems with this approach. Firstly, it may be difficult to determine the more general data structure which the service is to be written in terms of. Secondly, whilst the code is adaptable with respect to some kinds of change to the underlying data structure, it is not adaptable with respect to all changes in the data structure. In other words, complete adaptability is not covered. Indeed, complete adaptability is not possible, because it assumes that all assumptions that the code makes about the data structures that it uses are extracted out. This is similar in principle to open programming espoused by the reflection research community [Kiczales96a, Kiczales96b], in which a set of implementations of a particular aspect of the code are provided, which the user is able to choose from (see section 6.8). The similarity is characterised by the fact that both approaches rely on the use of an interface behind which a set of implementations (in the case of Aspect-Oriented Programming - AOP) or a set of specialised data structures (in the case of Adaptive Programming - AP) provides a choice for the user or software engineer. The important point to note is that the different implementations or data structures share a common interface. If a new implementation or data structure is required that doesn't satisfy the appropriate interface, ripple effects will occur.

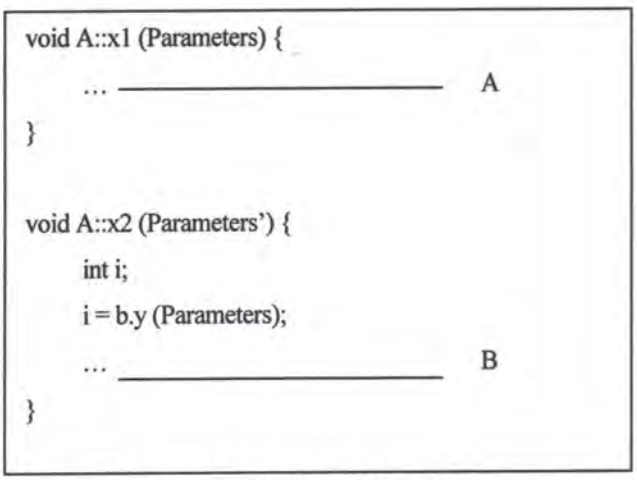


Figure 5 - Transformed Method

Propagation patterns are expressed in terms of paths through the class structure of the software as traversals from a start class to an end class. The propagation pattern traverses a path from the start class to the end class. Methods are related to particular classes and are called when that particular class has been reached on the path. The success of propagation patterns depends on the existence of a unique path from the start class to the end class. This is not always possible, since many paths may exist from the start class to the end class. Propagation patterns overcome this by allowing constraints to be specified, which are expressed in the form of particular classes that must be avoided. Thus, whilst some changes in class structure can be automatically adapted to by propagation patterns, there are some changes in the class structure that can't be adapted to and inevitably result in changes to the constraints. In addition, these constraints are class dependent, so that the class-dependence is shifted from propagation patterns to their constraints. It is unclear whether, after the class change, it is possible for the end class of the path to be un-reachable. It is also unclear whether or not this has been

proven, and whether it's possible for changes in the class structure to result in methods being executed in the wrong order. Propagation patterns require all method calls to be expressed at the propagation pattern level i.e. methods calling methods is not allowed because then changes in class structure could potentially affect the methods.

Demeter discounts the possibility of classes having recursively-defined elements such as linked list nodes. This simplifies the work but prevents the modelling of important data structures such as linked lists and queues which, even though they can be represented using other means (e.g. arrays), nevertheless discounts the use of a useful construction for modelling.

In addition, Demeter doesn't deal with evolution of classes with respect to methods i.e. how adding a method, removing a method and changing a method's interface affects the consistency of the software. It also doesn't deal with adaptation of methods to changes in methods on which they depend.

Demeter assumes that the control aspect of the code (propagation patterns) can be separated from the underlying class structure. This depends on the context dependence of the control element. For example, a filter control pattern (one that accepts input, filters it and then outputs it) is fairly context independent. It can be used on a number of different class structures conforming to fairly un-restrictive criteria, which only require the use of an ordered sequence of input service, filter service and output service. Other control elements can be very context dependent. The context dependence of a control element is also linked to its reusability – highly context dependent control elements are restricted to a small set of class structures and are therefore not very reusable. Context independent control elements are more reusable because they can inherently be used in many different situations. In the case of Demeter, it is arguable whether or not propagation patterns can be used to successfully capture the control aspects of all software.

There are a number of disadvantages with the Demeter approach:

- Adaptive Programming (AP) views adaptation of behaviour with respect to changes in contexts, but fails to address other forms of adaptability. For example, the adaptability of data conversions with respect to changes in their contexts, in this case the DEMs that are used by the data conversions;
- Demeter is limited to multi-object behaviour because the main construct (propagation patterns) is inherently multi-object and therefore isn't easily applied to individual object methods. An additional constraint is that the propagation patterns must be fairly generic so that they can be coupled to many different class models, in order to be useful. Hence, a propagation pattern which is fairly context dependent is not very useful because it can only be applied to a restricted number of class models, thereby removing any advantages of creating it in the first place in order to utilise its adaptability to changes in the class model;
- In propagation patterns there is a need to specify constraints i.e. nodes/objects not to visit, which thereby become potential targets of ripple effects. Hence, propagation patterns aren't completely adaptable because there may be a need to change the constraints;
- A traversal is a sequence of classes to traverse as part of a propagation pattern. A traversal is expressed as a sequence of visitor methods which constitute a call of a particular classes method, which implies that no

conditionals/loops are in the propagation pattern but are restricted to be in the methods themselves. This lessens the modelling power of propagation patterns;

- Propagation patterns are limited to the modelling of "stand-alone" behaviour; that is, behaviour which is not tied to any particular class hierarchy and is, to some degree, reusable. This restricts the scope of application of propagation patterns to standalone, multi-class behaviour. In addition, they can't be used to model complex behaviour consisting of nested method calls, loops and conditionals.

An advantage of Demeter is that the graph traversal mechanism which it employs does not require the software engineer to choose the generic interface to the data structure, because the graph traversal mechanism is inherently generic enough for any object-oriented data structure.

6.6 Rule-Based Software Architectures and Blackboards

Ward argues that using a rule-based system to model the real world has two problems:

- The knowledge elicitation problem: the modelling of domain knowledge as a set of rules may not be straightforward or even possible;
- The use of rules may not be simple [Ward94a].

Ward's alternative argument is for the use of domain-specific languages as an approach to software design. However, as Kanada points out, rule-based systems are very suitable for writing incomplete programs [Kanada94a] because P- and E-type software is continuously prone to change (Lehman's First Law of Program Evolution) and therefore all software is essentially incomplete because it never meets its users' requirements. The use of pre-conditions in rule-based systems doesn't build in any explicit control mechanisms like traditional hard-coded control architectures do. This, however, is also their disadvantage because control does often need to be hard-coded in.

Blackboard architectures are an important general problem-solving architectural type based on the use of knowledge sources, which are particular kinds of rules that encapsulate problem-solving knowledge. A **blackboard** consists of knowledge sources, which encapsulate procedural knowledge about how to perform a particular task. Each knowledge source has a trigger condition that must be satisfied before the knowledge source can be triggered and a pre-condition that must be satisfied before a triggered knowledge source can be executed. Of course, details vary, but the general mechanism is the same in all blackboard architectures. Specific details here refer to BBK (a C++ implementation of BB1 [Brownston95a]).

Blackboard architectures such as BB1 [Wolverton94a] model both **domain knowledge** and **control knowledge** using the same knowledge-source mechanism. An explicit control thread in the form of a meta controller/scheduler controls how both sets of knowledge interact. In this way, domain knowledge and control knowledge compete for scheduling. The meta-controller, unlike in traditional software, is fairly basic, consisting only of a loop that chooses to schedule knowledge sources based on their trigger condition and pre-condition. Blackboards are therefore opportunistic and are typically used in classification problems, in data-oriented applications or in problem-solving applications that involve the

interpretation of data. As a candidate for flexible software they do however suffer from program comprehension problems when the changes are non-configuration type changes. For changes of this type, new concepts will be required (since the required change can't be expressed in terms of the existing concepts) and program comprehension is therefore a necessity since the software engineer must determine how the new concepts relate to the existing concepts. In order to do this, the software engineer must comprehend the existing concepts.

Blackboard architectures are data-driven. This may conflict with requirements such as the need for an interrupt-driven model. Blackboards, like any other software architecture, are appropriate for certain requirements/software systems and not for others. The characteristics of the domain that make the use of a blackboard software architecture appropriate are:

- Data-driven control;
- De-centralised control i.e. control is emergent;
- The need for expression of functionality in terms of condition-action rules.

Unless these characteristics are met by the domain, the use of a blackboard software architecture will probably result in conflicts with requirements of the system as a whole.

In particular, blackboards aren't very good at modelling control applications. They're intended for data-driven applications. Pre-condition-based architectures in general can't be used for control-driven applications, because the definition of a control architecture/application implies that there is a control thread that explicitly calls particular services. In a data-driven architecture/application, there is no control thread and services are called based on the environment. However, pre-conditions can still be used to model services in a context-independent manner. The pre-condition consists of two parts:

- Data/parameters pre-condition;
- Environment pre-condition.

For data-driven services, both pre-conditions are important, for control-driven services, only the data/parameters pre-condition is important. The environment pre-condition can be simply set to true, indicating that the service can be *potentially* applied in any environmental condition. This doesn't mean that it is valid semantically in a particular application to use a service in any environmental context, but that this choice is only constrained by the control element of the software. There is a trade-off between having a complex control construct in which all the control knowledge is centralised in the control construct with no environment pre-conditions on the services, and a simpler control construct which consists of meta-control and the rest of the control is delegated to separate services which have pre-conditions indicating when they can be used.

For example, consider a compiler with a shared repository architecture. There are two choices:

- Have a complex control element that contains all the control knowledge for how to compile a file;

- Have a simpler control element that consists of read token and update shared repository, with pre-conditions on the services representing parsing, code generation etc. indicating the state of the shared data that triggers these services. New services can then be added quite easily and checked for conflicts with existing pre-conditions.

Blackboards are used in problem solving like in the "travelling salesman" or "greatest common divisor" problems, in which the programmer specifies the desired solution/result either directly or in terms of constraints. For example, "greatest common divisor" specifies that the result must divide the two input numbers and be the greatest number to do so. In some cases, not only the result is important (the entity that satisfies the specified constraints) but how the program got to the result.

In order to be able to use blackboard architecture in software in general, there is a need for a way to be able to specify the desired requirements in terms of a set of constraints. This may not always be possible, or the constraints may be very complex.

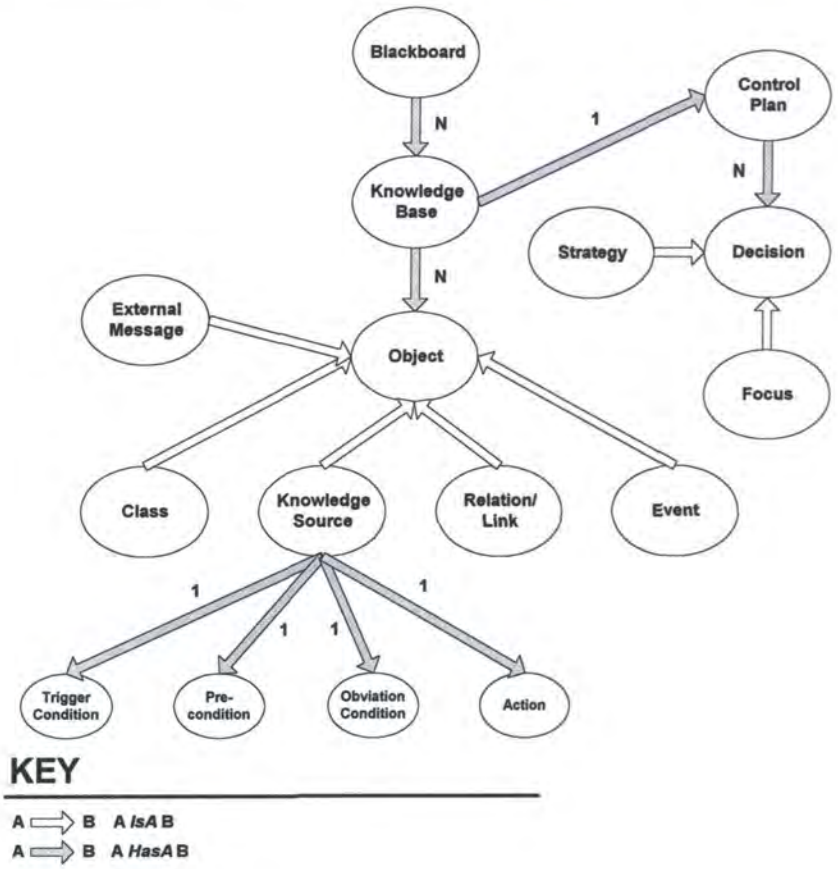


Figure 6 - BB1/BBK Architecture

In a blackboard, domain knowledge and control knowledge are treated the same i.e. are both represented as knowledge sources. This allows, amongst other things, domain and control knowledge to be reflected upon. The control plan in BB1 consists of a set of related objects of type either strategy or focus. The focus objects take a knowledge source as parameter and return a rating for that knowledge source, which is then used in conflict resolution for scheduling purposes. Strategy objects control the switching on and off of certain focus objects under their control, thereby providing some form of control over the focus objects and in turn the control aspect of the blackboard. In essence, they provide a limited planning mechanism, "…the strategy objects constitute the skeletal plan." [Wolverton94a]

Trigger conditions are evaluated in terms of trigger events, whereas preconditions are evaluated in terms of the blackboard as a whole. As a result, blackboards are essentially event- and data-driven, with control provided as and when required. This is advantageous as a generic architecture in which it is possible to model purely data-oriented applications, purely control-oriented applications and those in between these two extremes. The data-driven nature of the meta-controller and pre-conditions, coupled with the ability to model control as Knowledge Sources allows this. But, blackboards aren't very good at expressing well-defined control tasks. For example, a user wants to be able to read a file, sort it and then output it on the screen. The control aspect of this is well-defined. A blackboard approach is contrived.

However, there may be problems with constructing pre-conditions due to the potential heterogeneity of concepts used in the pre-conditions in the existing software system. The software engineer needs to match concepts used in the pre-condition with those used in the software system in order to ensure that the pre-condition will at least be compatible with the existing software.

A potential problem with blackboard software architectures is that, due to the de-centralised nature of their control, pre-conditions need to be checked for every triggered KS. The number of triggered KS's may be quite small for most applications, and hence the number of pre-conditions to check will be quite small. Checking a large number of pre-conditions can be eased by the following algorithm.

If all data structures are modelled as an ADT or class, then there are two types of method:

- Those methods that alter some part of the data structure – put operations;
- Those methods that don't alter any part of the data structure – get operations.

Let each data structure possess an id. Let "Updated" be a set that contains the data structure ids of all data structures updated during the execution phase of KS's on the blackboard. Let "Mappings" be a hash table (or associative array) that maps data structure ids to the KS's whose pre-conditions use that data structure. Let each method that uses a put operation also include a call to an operation that updates the "Updated" set with the id of the data structure to which the method belongs. Let "Checked" be a set that contains those KSs that have been checked for triggering by the algorithm. Then, the triggering phase of the blackboard meta-control process consists of the following:

```
Checked = {}          // For each data structure updated during the previous execution phase, get
                      // KSes that use them and match them to triggered KSes
Foreach DataID (Updated) {
    Vector KSes = Mappings.get (DataID);      // A Vector is a dynamic array
    Foreach KS (KSes) {
        If (not Checked.contains (KS)) {
                preconditionCheck (KS);
                Checked.add (KS);
            }
    }
}
```

There are a number of factors that influence the efficiency of this algorithm:

- The number of KSs. The larger the better, basically (a smaller number of KSs will not benefit from the algorithm because they can be checked quickly);
- The size of the KSs. The larger the better, again, because re-computing large ones un-necessarily (because the data they use hasn't been changed by any operations on the previous execution phase) will take a long time.

A blackboard approach provides a basis for glue-less components, in which individual elements of the software (such as knowledge sources) are individual entities and there is no glue, or explicit control element, to tie them together in any particular configuration. Such configurations emerge at run-time as part of the pattern-directed run-time nature of the blackboard architecture. This allows new software elements to be slotted in (with their appropriate pre-conditions and trigger conditions) and used whenever appropriate (i.e. when their pre-conditions and trigger conditions are met). This is flexible but has a number or problems for software maintenance and evolution. Firstly, it doesn't help in system comprehension by allowing the software maintainer to determine if a new requirement is satisfied by the existing software elements. Secondly, control evolution is a potential problem. Control is implemented in terms of the blackboard using priorities, which are required only when a number of knowledge sources are competing to be executed at any one time. Whilst priorities are implemented in terms of the blackboard itself (thereby providing a homogeneous mechanism for describing both software elements and priorities), some comprehension phase would be required in order to move from the current control mechanism to the required control mechanism. This is because the current control mechanism must be understood in order to adapt it to the new control mechanism that would satisfy the new requirements.

6.7 Reflection (Self-Modelling)

The concept of reflection (or self-modelling) was initially explored by John Macarthy in his work on LISP. To date, it has mostly been applied to software languages, where the underlying mechanisms of the language, such as messages, classes, methods etc., are reified, meaning that they are represented within the language so that they can be manipulated. A major aim of reflection has been to allow software engineers to change the semantics of aspects of a language and provide a well-defined interface for doing this.

Brian Cantwell Smith, in the early 1980's, applied the idea of reflection to procedural programming languages [Smith82a]. Maes has applied it to object-oriented languages [Maes87b]. Since then, different applications of the technique and different types of reflection have emerged, from computational reflection [Maes87b], to meta-programming [Denno96a, Kiczales91a] and software architectural reflection [Cazzola97a]. A software system using reflection has at least two levels of processing:

- Base level, dealing with the domain of processing;
- Meta level, dealing with strategies based on the base level i.e. with how base level processing proceeds;

although there may be more than one meta level.

In most cases, the approach involves constructing a hierarchical tower (called a reflective tower) consisting of increasingly more abstract layers that extend from a base level through successive meta levels. Each meta level models the layer below. The different applications of reflection differ in the following ways:

- The number of meta-levels;
- The form of the meta levels;
- What the meta levels model, and what this information is used for;
- How much is reified in the meta levels.

Each level contains a model of the concepts used within that level. An important aspect of this is how the meta level can exert control over the base level in order to effect changes in its topology or strategy. Cazzola et al suggest the use of a state machine-based approach to modelling in each layer, which then leads on naturally to the use of state machine operators as a way for a layer to make changes to the layer below [Cazzola97a].

Most reflective software systems to date, such as 3-LISP [Smith82a], SOAR [Laird86a], META-PROLOG [Bowen86a] etc., use only one meta level. The form of the meta level refers to the constructs that are used in the meta level. The form of both the base level and all the meta levels was expressed in terms of a language, of which there are two types:

- Declarative, and;
- Procedural [Maes87a].

Declarative reflection languages are arguably the more powerful, but procedural languages are more flexible.

Reflective approaches mainly differ in how this reflective tower is utilised. For example, in meta-programming, the tower is essentially a way of modularising the code and thereby localising changes to as few layers as possible. Most of the work in this area has been concerned with using meta layers as ways of introducing management aspects to the code such as location transparency [Stroud92a].

However, all reflective approaches have a number of things in common:

- The ability to model aspects of the code;
- The ability to change these models so that these changes result in changes in the actual code.

Both are important for the reason that, as with any modelling, the model is not an end in itself but is used in some higher level mechanism i.e. there is a reason for the modelling. In the case of reflection, there are three reasons for modelling:

- To add some "self-awareness" to the software, so that it can analyse what it is doing to some extent and then use this information to change its behaviour;
- To document the software, so providing a link between a higher level model of the software and how that model is actually implemented;

- To provide a way of allowing evolution to take place outside the space determined by the start model and the set of allowable operations by using meta models to extend the operations in a lower level [Cazzola97a]. Cazzola et al, for example, use a reflection approach at the software architecture level to model a distributed track control system for a railway [Cazzola97a]. The base layer consists of track components and connectors that connect only adjacent tracks, along with operations to add and remove tracks that preserve the constraints, so providing a space of topologies for the tracks. The meta layer deals with changes that are outside the space of topologies allowable in the base layer by allowing changes to the operations and constraints of the base layer.

An intuitive heuristic of particular importance that relates to the reflective aspect of the work is that of reifying[6] as much as possible in the reflective models, a point made by Maes in [Maes87a]. If something is reified then it can be reasoned about. This allows the software to reason about its own evolution. The identification of a rich set of software entities (see chapter 4) goes some way to addressing this heuristic.

Existing research into computational reflection has focused on altering the *existing* functionality of software by replacing an aspect of the software (e.g. a function) with another function at run time. This technique is used, for example, in LISP to alter the functionality of LISP itself [Maes87a].

As Maes points out, it is important to reify (model in the meta level) as much of the domain as possible [Maes87a]. This allows one to manipulate and reason about the reified component. The only way for a component to be manipulated and reasoned about is for it to be reified in the meta level. The reified components are represented at the meta level by other components. Their form is determined by the purpose or goal of the reflection. As Maes states:

> "These structures [reifying the lower-level components] include data representing entities and relations in the domain and a program prescribing how these data may be manipulated." [Maes87a]

In other words, the meta level includes:

- Meta-data *about* the base-level component that is being reified;
- Operations on this meta-data;
- Relationships between meta components and components in the layer below.

The last point is important: it means that the software engineer must provide operations that prescribe how the meta-data can be manipulated. The causal connection that exists between the base- and meta-levels then ensures that any changes to the meta-data percolate down to the base level.

Researchers have reified many different types of software entity. Maes, in her work on computational reflection which resulted in her Ph.D. thesis of 1987, reified object oriented software entities such as:

- Classes;

[6] Or, representing at the meta-level. An object in a base or meta level is represented in the next meta level.

- Inheritance Relationships;
- Object-object messages [Maes97a].

This reification allowed her to manipulate these constructs at run-time and change their semantics or behaviour.

Others have extended the work on reification of messages to allow interception of messages at run-time in order to provide some new behaviour [Maes87a]. Stroud describes the reification of operation calls in order to extend the functionality of a software system [Stroud92a]. He also points out that in order to be able to perform reflection, there must be a suitable interface, one that allows the reified component(s) to be manipulated [Stroud92a p100]. Another important point is the level of abstraction of the reified component(s). More powerful reflection can be obtained by reifying more abstract components of the underlying base domain level. Stroud cites as an example the UNIX operating system[7] and specifically the file system. Interception of operation calls at the device-specific layer of the file system (which is responsible for data storage and organisation) is used by NFS which intercepts name lookup and data transfer operations to build a stateless file server. The next higher device-independent layer is responsible for parsing path names and calling the device-specific layer with the appropriate parameters, which are dependent on the path names [Stroud92a p100].

In summary, reflection makes explicit in software that which is typically implicit and not manipulate-able. It gives the software engineer power over altering aspects of an implementation that are typically hard-coded. For example, reflection allows one to manipulate software architecture, an aspect of software that is typically implicit in software and not manipulate-able, or how inheritance is implemented. It also allows executing software to reflect upon itself, for example to determine previous events that have happened and which services have been executed.

The reflective tower consists of a base layer (the actual model to be changed) on top of which exists a set of meta layers, each one providing a higher level model of the one below. A meta level can be used either as a knowledge tool to allow the maintainer to understand the context of a change in the layer below, or as a trigger for change in the layer below. For the latter, changes in the meta layer automatically produce changes in the base layer.

Reflection also helps to tie down where changes have to be made, by providing a hierarchy to concepts modelled within the reflective tower. As already pointed out, a layer provides a space of topologies/control strategies to the layer below. Any changes outside this space of topologies means that the change has to shift to the layer above [Cazzola97a p8]. In their case study, Cazzola et al use the example of a distributed railroad track control system [Cazzola97a], whose architecture consists of track components each controlled by a software controller. The connectors connect only adjacent tracks and the protocol is such that tracks must be reserved before they can be used by a train. The base layer consists of operations on tracks such as add and remove a track, whilst preserving certain constraints in the meta layer. For example, the meta layer may contain in its model information that constrains the topology of the track to a star-shape. Changes that invalidate this constraint must be shifted up to meta layer and this information changed to allow other topologies to be used.

[7] UNIX is a registered trademark of AT&T Bell Labs.

Reflection, however, doesn't provide a complete solution to evolution because it *generally* only allows the behaviour or semantics of an *existing* aspect of the software to be changed. However, reflection can be used in other areas. For example, message reflection/interception allows the semantics of the message to be re-implemented by providing extra behaviour. This means that reflection provides no real advantage over existing ways of changing code, just a different way of doing it. However, reflection coupled with a richer set of software entities may help in this regard. This richer set of software entities could include goal software entities that describe a goal of the software and map it to the code that carries out the goal. Reflection could provide a way for the semantics of the goal to be altered at run-time, for example.

Meta levels can be used to structure the evolution operators. The higher up the meta-level hierarchy one goes the more abstract the evolution operators become. For example, for a software architecture entity, the meta level might contain evolution operations within the capabilities of the particular architecture chosen. At higher meta levels, the evolution operators would be more abstract and more powerful, allowing evolution to occur that drastically changes the architecture. Obviously, this has to be constrained by rules so that the model of the software under the existing architecture doesn't change. Otherwise, changing the architecture would result in changing the behaviour of the software.

Work on reflection attempts to open up languages by allowing the user of the language to customise the language in a well-disciplined manner. In this way, specific assumptions are lessened, because the implementation of a particular aspect of a language can be chosen by the end-user. This implementation is traditionally not customisable in traditional languages, leading to built-in assumptions.

Reflection is only useful for re-configuration evolution because it's generally used to change the semantics of existing concepts in a software language. It doesn't deal with integration evolution. Viewed in terms of the framework of software entities introduced in chapter 5, reflection changes the implementation of a software entity:

Software-Entity →implements→ Entity-Implementation

The semantics of "Software-Entity" are changed by changing the implementation relationship. For example, for the "Message Software Entity" implementation shown in Figure 7, the implementation of the message software entity can be changed by changing the implementation relation. Typically, approaches to reflection adopt an object-oriented approach to structuring the reflective model, so that changes can be made using inheritance. The problem with an object-oriented modelling approach, and therefore with reflection as a form of software evolution, is the inflexible nature of the interfaces. Hence, software evolution is restricted to re-implementing a capability using existing interfaces.

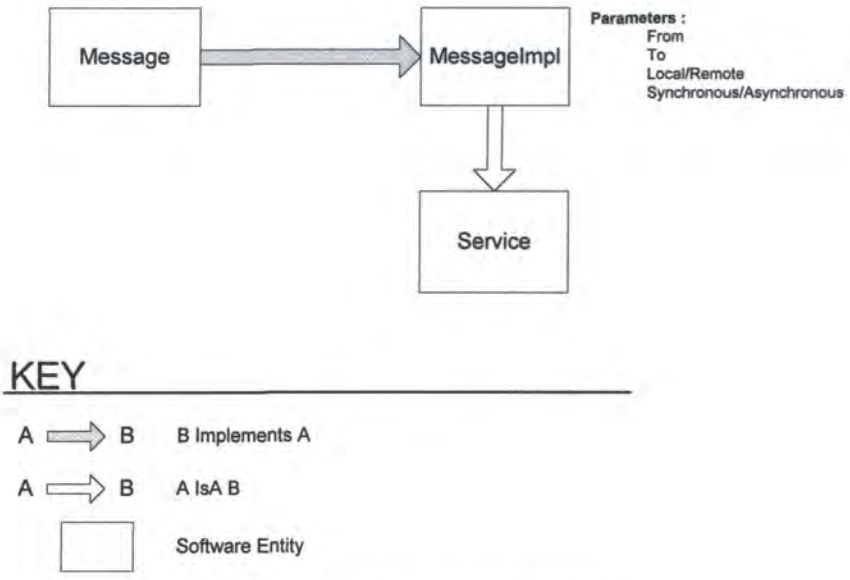


Figure 7 - Reflection and Software Evolution

In summary, reflection provides a protocol for making changes to *existing* aspects of the language or software model which is being reflected upon. The protocol can take the form of an object-oriented model, so that changes can be effected using inheritance. Reflection is no good for integration evolution, where changes occur outside the well-defined structural model provided by the reflective model. Reflection is like any model, consisting of components with interfaces. Changes outside these interfaces fall outside the protocol provided by the reflective model, inevitably resulting in integration evolution.

The protocol provides a set of hooks for particular types of changes at chosen levels of abstraction, by allowing the user to change the implementation of particular aspects of the software. In this way, reflective approaches to software evolution are similar to open implementation approaches, in which particular types of changes are made more easy to perform, for example, changes to the implementation of programming language function calls through the reification of messages.

What is the difference between a reflective software entity and a non-reflective software entity? Both use other software entities. However, whereas non-reflective software entities use extensional software entities, reflective software entities use intensional software entities i.e. software entities which may not currently exist but, when they do, will conform to an existing meta interface.

Reflection is good at providing for extensibility because a meta-level, if designed well, provides an interface for evolving a particular (set of) aspect(s) of a software system. The level of extensibility is dependant on the number of meta levels in the model. The more meta-levels, the more extensibility is possible, because higher levels can provide extensibility for lower levels and the levels can modularise the evolution space of the software system. A major limitation is that reflection doesn't deal well with changes which break the existing interfaces expressed within the base and meta levels.

So, in summary, reflection provides a protocol for making changes to the *implementation* of an *existing* software entity or concept.

6.8 Open Implementations and Evolution spaces

Open implementations are an approach to overcoming the difficulties posed by the black box abstraction principle, which separates the functionality (the interface) from the implementation (the encapsulated or hidden part of the component). The idea of open implementations grew from, amongst other things, performance problems when components were reused for tasks that they weren't originally designed for, and for which their in-built assumptions were invalidated. The primary conclusion of open implementation research is that it's difficult to hide all implementation decisions from the user or client of a component. The primary difficulty is in designing the interface so that the client of the component is able to make design decisions through the interface. Figure 8 shows how open implementation works. The server is a grey box in which the implementation isn't fully hidden. There are two interfaces to servers:

- Usual (functional) interface;
- Open implementation interface – a documented interface in the form of more parameters to the server that allows the client to choose from a set of implementations.

Open implementations are related to computational reflection, since both allow the implementation of an existing concept in the software to be changed. Reflection allows the software engineer to design the meta levels of the software and thereby control how individual concepts can be adapted. In essence, reflection provides a mechanism for changing the base level concepts. Open implementations, in contrast, provide a limited number of implementations of a concept to users of the concept, accessed through a secondary interface.

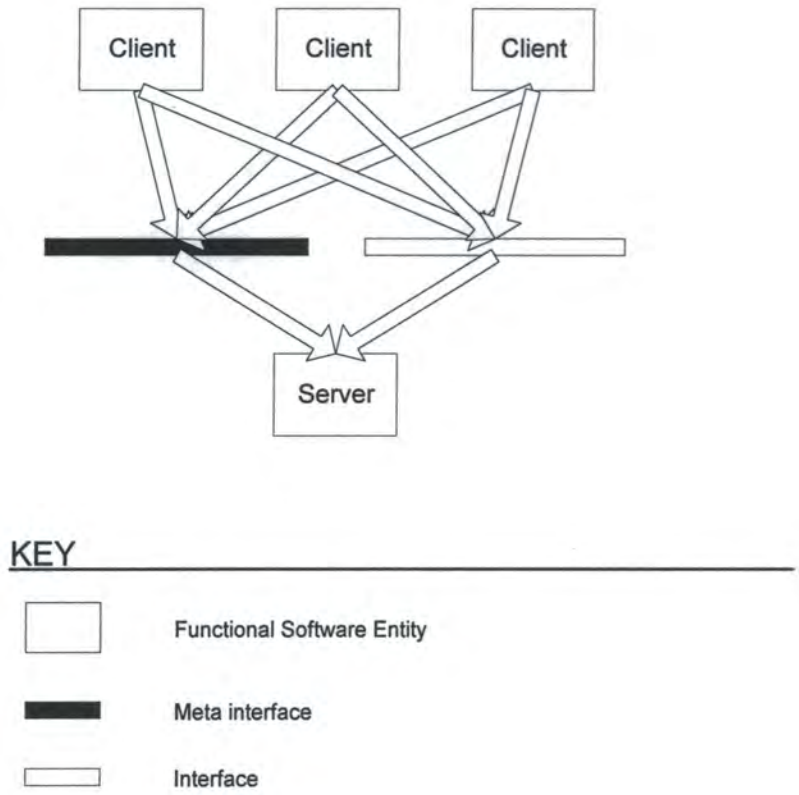


Figure 8 - Server Interfaces

Kiczales et al propose defining/identifying a set of object types and operations on them, which can support not just a single behaviour, but a space or region of behaviours which they call a protocol [Kiczales91a]. A default behaviour, or point in the region of behaviours, is defined along with an interface that allows one to switch between points in the protocol or behaviour space.

Open implementations allow any clients of an abstraction (or component) to choose the implementation of the abstraction. In effect, an evolution space is provided for the component, albeit a built-in and restricted evolution space. However, adaptations are restricted to the abstraction's interface which prevents ripple effects percolating to clients of the component. The most general interface required to cover all adaptations is chosen, so that changes to the interface don't need to occur.

An important issue in open implementation is the binding time associated with the software entities to which open implementation is being applied, and the level of abstraction chosen at which to implement open implementation. A low level of abstraction implies a limited choice of implementations. For example, consider the decision of where to apply open implementation given two choices:

- A sort entity;
- A bubblesort entity.

The sort entity is more abstract than the bubblesort entity because the bubblesort entity is an implementation of the sort entity. Applying open implementation for the bubblesort entity doesn't make sense because there is only one

implementation of bubblesort and therefore very little to parameterise its evolution (or implementation) space. In contrast, the sort entity has a larger evolution (or implementation) space, consisting of all existing and potential sort implementations, parameterised along a number of axes, including memory requirements and speed or execution. Hence, open implementation could be applied to the sort entity, and parameterised along these axes. The binding would be very late, in this case at run-time because the decision of which sort implementation to choose is based on the input set and the memory and speed requirements. This example also provides an example of the removal of assumptions, their transformation from implicit assumptions to explicit parameters. In this case, memory and speed assumptions are transformed into parameters.

Open implementations have some similarity to software which provides hooks that allow software entities to be easily integrated with an existing software system. For example, Netscape Navigator provides a hook mechanism whereby plug-ins can be plugged-in to provide a way to interpret specific data. In this way, a new type of data arriving at Netscape Navigator that can't be displayed can be passed on to a plug-in that has registered itself to display that data. The main constraint is that the hooks have associated interfaces which the new software entities must adhere to. In the case of Netscape Navigator plug-ins, the interface is in the form of an input mechanism on the plug-in that accepts data in a pre-defined format.

Both open implementations [Kiczales96a, Kiczales96b] and evolution spaces [Cazzola97a] are re-configuration approaches to evolution, which adopt similar stances. Both approaches provide a set of implementations, an evolution space, for a particular well-defined interface that itself provides a set of parameters that allow one to choose a point in the evolution space, or a particular implementation. Both approaches emphasise the need for the software engineer to choose the aspects of the code to be reified (modelled directly and thereby given an interface), and then provide an appropriate interface and evolution space for these reified entities.

Re-configuration approaches like this can help with a sub-type of re-configuration evolution called extension evolution, in which new capabilities are specialisations of *known* capabilities/concepts in the software, in which case they are specialisations of these capabilities' interfaces. In this case, the changes have been provided for indirectly because, even though they may not have been predicted, the parts of the code of which they are specialisations, by their very presence in the software, implicitly predicts their future use.

There are a number of problems with open implementations:

- Choosing which implementation decisions to make public;
- Choosing the most general parameter set. That is, choosing the most general number of parameters and the most general type for each parameter may be non-intuitive;
- They only target part of the problems of evolution by considering only a particular type of change i.e. those changes which change the algorithm for a particular abstraction. Similarly, meta protocols have only been targeted at particular types of changes, such as the introduction of fault tolerance into a software system, thereby essentially predicting the change to be made which is contrary to the assumption of this thesis that changes in general can't be predicted;

- Their lack of support for integration evolution. However, integration evolution implies the integration of *new* capabilities with the existing software, and it is generally unknown what the relationship is between these new capabilities and existing capabilities so that the software engineer must make changes which are arguably more complex than changes made during re-configuration evolution.

6.9 Hooks for Software Evolution

Hooks can be provided in software in order to ease the integration of new capabilities into the software. An example of this includes Netscape Navigator's plug-ins mechanism, whereby external applications can be registered with Netscape Navigator, to be executed to display a set of particular MIME (Multi-purpose Internet Multimedia Extensions, a syntax that provides a way to describe what the data in a data stream is [Freed96a]. For example, a particular graphics format, or a word processing format) types that Navigator itself is unable to display. Another example is provided by UNIX and other operating systems, which de-couple the core kernel from hardware dependencies by the use of device drivers. Device drivers are hardware-dependent implementations that provide access to the functionality of the hardware they control. The design of the device driver framework provides a means for integrating new device drivers into the operating system and hence allowing access to new hardware.

The interface provided by the hook is important. Some hooks provide a mechanism for integrating new capabilities that implement a particular concept, and so are inherently constrained. The Netscape plug-in example above is an example of this because the plug-ins display particular types of content which Netscape Navigator itself isn't able to parse, and hence must implement an interface that allows Netscape Navigator to request the plug-in to display the content. Other hooks may provide a more generic interface and thereby may not put any major constraints on the capabilities that use the hook. The onus is then on the capability to correctly parse any requests received through the hook. An example of this is the device driver framework of UNIX, which provides a means to add in new device drivers without having to re-compile the kernel. The hook in this case is in the form of a kernel-device-driver interface provided by the "ioctl" system call. This system call provides a communication pipeline between user programs and the device drivers. The pipeline is generic, so that invalid requests may be sent to particular device drivers.

Hooks are related to extensibility because a hook interface provides a mechanism for "slotting in" appropriate components which conform semantically to the interface. Classes and polymorphism in object-oriented languages allow a software engineer to design an abstract class, for which subclasses provide more concrete behaviour. The abstract class is the hook and the subclasses implement the hook interface. Polymorphism ensures that some aspects of the hook interface are semantically valid in the subclasses, but don't "check" whether the behaviour of the sub-classes is appropriate for the hook interface.

The major challenge for integration evolution however is in integrating new capabilities that don't conform to existing interfaces in the software.

6.10 Structural Modelling

The structural modelling approach [Abowd93a, Chastek96a] minimises the number of data and control connections between components through the use of indirect data and control relationships, which increase the modularity of software by de-coupling the computation from decisions about how data and control are passed. A broker architecture is utilised, in which components communicate data and pass control to other components through a broker. This minimises direct connections, so decreasing the coupling and increasing the integrability of the architecture:

> "...the integration problem has been reduced to a problem that is linear, rather than exponential, in the number of components." [Bass98a p325]

Even though this approach increases the complexity of the brokers every project that has used structural modelling has reported "...easy, smooth integration." [Bass98a p325]. It is unclear, however, if this is limited to just flight simulators, which is where structural modelling originated.

Integrability is a measure of the ease of integrating new capabilities with existing capabilities. It is an important measure for integration software evolution, which reports on how easy it is to integrate new capabilities with existing capabilities. As Bass et al report, the structural modelling approach improves the integrability of the architecture. Integration is eased in the case where the interface required by the new capability is compatible with the interface of the existing capability with which it is to be integrated. Even structural modelling will not ease integrability if the interfaces are incompatible, which is another example of built in design decisions and assumptions conflicting with new design decisions.

7 Summary, Discussion and Conclusion

This chapter has explored a number of approaches to the various aspects of software evolution identified in chapter 2.

The problems with the approaches described above include:

- Hooks – future capabilities must conform to a specific interface;
- Reflection – can deal with re-configuration evolution and partial integration evolution, only if the changes are within the interfaces of the reified entity;
- Open implementation – can only evolve within a pre-defined space.

The main conclusion about existing product-based approaches to software evolution (as opposed to process-based approaches) is that they are re-configuration evolution approaches, in which future requirements have essentially been guessed and measures built in to provide a re-configuration that chooses an alternative implementation through an "evolution space" of such implementations, together with a default implementation.

The evolution space concept is applicable at any level of abstraction, so that the basic constructs of a software language provide a potential evolution space. The problem with this is that they're low level. Open implementations extol the

virtues of re-configuration evolution and evolution spaces by placing the level of abstraction at which changes in implementation can be made at a much higher level. Kiczales gives the example of a spreadsheet application built as a two-level hierarchy, with the spreadsheet functions at the higher second level and the cell-implementation at the lower level [Kiczales92a]. The key to open implementations and re-configuration evolution is in choosing an interface to the lower level, creating a set of implementations for that level, and choosing a set of parameters that allow one to re-configure/choose the implementation required at the lower level. These parameters must be general enough to cover the assumptions made in the different implementations.

First of all, the new requirements must be mapped to changes in the software, which are dependant on *both* the existing requirements (which are encapsulated in the software before evolution) *and* the new requirements because new requirements may conflict with existing requirements. Hence, there are two aspects to software evolution:

1. **Primary evolution**, which emphasises evolution resulting in the implementation of requirements which *extend* an existing software system;
2. **Secondary evolution**, which emphasises evolution as a result of conflicts in requirements, revocation of design decisions etc and usually results in ripple effects.

Both types of software evolution are of concern to this thesis, which aims to make all types of change easier to make through the improvement of the evolveability characteristics of software. This includes:

- **Flexibility**, which aims to help with both types of evolution by improving the ease with which existing requirements and design decisions can be revoked;
- **Adaptability**, which aims to help with secondary evolution by limiting ripple effects, where possible;
- **Management of ripple effects**, which aims to help with secondary evolution by allowing *ripple effect types* to be determined;
- **Extensibility**, which aims to help with both types of evolution, by introducing architectural features that permit the modelling of extensible abstractions *directly* in the modelling framework of SEvEn.

Another classification of software evolution types is in terms of re-configuration evolution and integration evolution, as discussed chapter 2 section 1. The main challenge in software evolution lies in integration evolution, in which new capabilities are required that must be integrated with existing capabilities. Most current approaches to easing software evolution, described in this chapter, rely on not breaking interfaces by making the interfaces as generic and abstract (adaptable) as possible. Furthermore, not all types of evolveability are covered.

An analysis of the evolveability of the approaches described in this chapter, along with other common software architectures, languages and models, will be discussed in chapter 8.

Chapter 4

A Conceptual Framework for Improved Software Evolveability

1 Introduction

Chapter 2 discussed software evolution in general, identifying what it is, how it is performed and what the problems are. Chapter 3 explored various approaches to software evolution. The rest of the thesis focuses on the specific software evolution problem of evolveability, which is concerned with the flexibility and adaptability of software. This is approached by first identifying a set of software entities with more flexibility and adaptability than existing software entities (chapter 5), and exploring their improved evolveability (chapters 6, 7 and 8). The resulting conceptual framework for software evolveability is called SEvEn.

Software engineering principles with respect to increased modularity, laid down by Parnas in the 1970's, aim for a structured model, whereby design decisions are hidden behind interfaces [Parnas72a]. The reasoning here is that design decisions for a particular capability (where capability is used in its broadest possible sense to include any abstraction used in a software system, from functionality to data structures and architectural patterns) may change (evolve) and the interface will protect clients of the capability from being affected by changes. This approach is fine for changes that can be predicted, but can't cope with software entities that haven't been encapsulated by an interface[1]. Parnas' original influential paper succeeded in identifying that modularity and the appropriate use of information hiding were important for encapsulating the effects of change, but research has since failed to capitalise on this observation and develop strategies for increasing the modularity of software.

There are two approaches to overcoming the problems caused by ripple effects:

- **Recovery**, in which ripple effects are recovered from after they have occurred. This comprises most of the research to date;
- **Prevention or reduction**, and containment through the limiting of assumptions and the use of adaptive software entities. In addition., the use of modularisation to increase the encapsulation of specific aspects of the software and, in particular, to improve the encapsulation of aspects of software that may change. This encapsulation hides specific details that, when changed, don't affect other parts of the software. The problem with current software is that it

[1] It would be interesting to analyse the ratio between encapsulated design decisions and non-encapsulated design decisions in software. The author is not aware of any existing research in this area. Of course, encapsulation here is with respect to future requirements which can't be predicted. Therefore, such a measure may be difficult to determine, so that it would have to be done with respect to *particular* new requirements.

focuses on the wrong modularisations. Modularisation is targeted mainly at modules and classes. We need to target it at more abstract levels, at software entities which cover a wider range of aspects of the software. For example, control software entities and code structure software entities have an inherent dependency between them such that when the class structure changes, the control aspect is affected (ripple effect type). Propagation patterns attempt to overcome this kind of ripple effect by de-coupling the control from the underlying class structure [Lopes94a]. This discussion recognises the fact that there are types of ripple effect at different levels of abstraction in the software that need to be addressed, and which aren't covered by existing software languages, architectures and models. For example, the dependencies between data mappings and data that can cause ripple effects are typically not modelled explicitly, so that it's difficult to determine when they occur in traditional software systems.

The emphasis in this thesis is on limiting the assumptions made, in order to improve the evolveability of software. The main thrust involves limiting the assumptions that the software entities described in chapter 5 make of those software entities on which they depend. The dependencies of each software entity identified in chapter 5 are determined in order to aid in this, and strategies developed for improving the adaptability of these software entities. Where this isn't possible because assumptions can't be extracted out, the types of ripple effect can be determined and measures built in to identify them and control their propagation.

Figure 2 depicts the interplay between client requirements[2] and servers that implement requirements. The figure is structured to indicate that there is a separation between the requirements that a client wants and the requirements that a server implements. (a) shows the situation before evolution in which the server implements the client's requirement, (b) shows the situation after evolution. Reqc, Reqc', Reqs and Reqs' are all sets of requirements. The relationship between them is important in determining both:

- The ability of the server to satisfy the requirements of the client;
- The potential adaptability of the client with respect to the server, and of the server with respect to the client.

Changing a requirements set is accomplished by applying any combination of the operators:

- Add a new requirement, with no conflicts with existing requirements;
- Remove an existing requirement.

The combinations of these operators determines the relationship between the requirements sets before and after evolution, as shown in Figure 1. As can be seen in the figure, Req' $\supset$ Req means that requirement set Req' extends (is a superset of) requirement set Req (without invalidating any part of Req through conflicts – conflicts are resolved outside of this model through removal of old requirements and replacement with the new conflicting requirement). Req' $\subset$ Req means that requirement set Req' specialises (is a subset of) requirement set Req (this is true even if all requirements from

[2] The requirements may be expressed in a requirements language, a domain language or some form of specification language. The exact form of the requirements is not of concern here, merely that requirements can be compared in the model chosen.

Req are removed, since set theory tells us that $\varnothing \subset X$, for any set X). Req' $\subset$ Req means that requirement set Req' extends a subset of requirement set Req, again with no conflicts. A special case of this is when all requirements from Req have been removed so that the intersection between Req' and Req is the empty set, $\varnothing$. Note that, before evolution, the conditions are $Req_S \supset Req_C$.

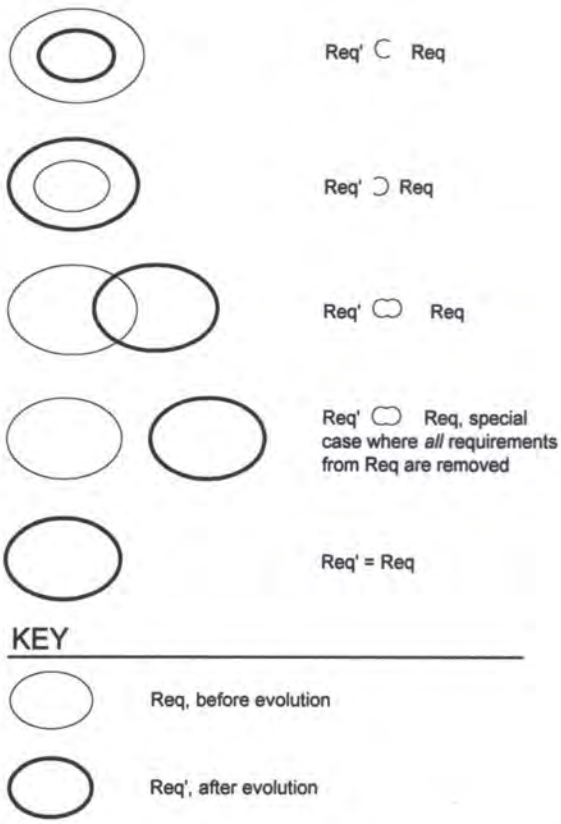


Figure 1 - Relationship Between Requirements Sets Before and After Evolution

The relationships in Figure 1 can be applied to:

1. Req_c and Req_c'
2. Req_s and Req_s'
3. Req_c' and Req_s'

in order to determine the evolution of client and server requirements in relation to each other, and thereby determine client and server adaptability. Note that 1 and 2 don't determine 3. Consider the situation in which $Req_s \supset Req_c$ before evolution (the server implements more requirements than the client requires), and $Req_c' \supset Req_c$, $Req_s' = Req_s$ after evolution (the client extends its requirements of the server, but the server still implements the same requirements). The server may still be able to satisfy the client's requirements, depending on the range of requirements it implemented before evolution, Req_s. Client and server adaptability are both dependent on the types of the client and server, and on the types of change to which the client and server need to adapt. For example, if the client is an FSE (functional software entity – see chapter 5)and the server is a DEM, then it should be expected that the FSE should be adaptable to changes of the form $Req_s' \supset Req_s$ in which the DEM is more generic but still semantically provides the original data requirements.

This, however, cannot be said if the server is replaced by an FSE, because adding functionality to an FSE will probably invalidate the requirements that the client makes of it.
An important rule can be stated at this point:

Rule: use the most abstract software entity interface possible, one that makes as few assumptions about its environment as possible.

Any software entity possesses an interface which is used by those other software entities that use it. This is also true of data, a type of software entity which isn't typically attributed with having an interface (except perhaps in object-oriented models). A data structure's interface is composed of the data entities which make up the data structure, along with the relationships between these data entities. This rule is ignored by many constructs used in modern programming languages, architectures and models. A good example is data structures which are often so specialised that when they change any functions that use them are invalidated. This is because the functions depend on the original specialised data structure, and hence inherit its assumptions. Another example is software architecture, which is often implicit in software, hides assumptions and on which FSEs depend. Changes in the architecture can affect the FSEs because of the implicit dependency between them and the original software architecture.

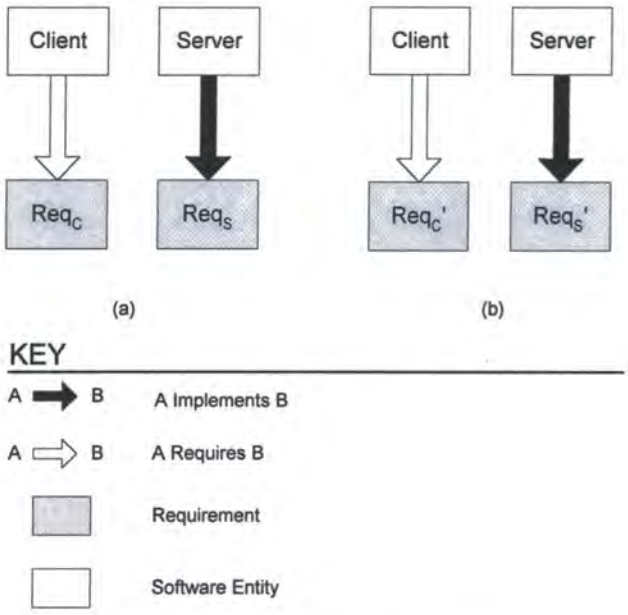


Figure 2 – Client Requirements and Servers

Hence, adaptability is dependent on:

- The type of client;
- The type of server;
- Whether adaptability is client with respect to server, or server with respect to client.

Adaptability is the ability of a client software entity to cope with changes in a server software entity on which it depends. The level of adaptability defines how many types of change in the server the client can adapt to without needing to be

changed itself. The definition of adaptability assumes that the client's requirements of the server stay the same, but the server itself may change. Changes outside the interface between the client and server will cause ripple effects, unless provisions are made to:

- Improve the adaptability of the software entities in software;
- Localise changes to software entities, which isn't always possible unless the interface is generic enough to support it.

This is approached in the main by the introduction of more interfaces, which implies an increased use of indirection. Interfaces are of two types:

- Passive, similar to existing programming language interfaces;
- Active, in which the interface is provided by a broker which performs some task and hence provides the interface with some functional capabilities.

2 SEvEn's Approach

One of the main aims of this thesis is to not rely on software entities that are developed in such a way as to make *particular* requirements easier to integrate with existing software entities. An example of this is a data structure software entity that allows the addition of a specific data element to be integrated easily. In effect, the design of the data structure predicts the addition of the new data entity and builds in measures to make the evolution easier. However, some predictive aspect can still be built in to software entities to allow certain types of change to be made more easily. Thus, it is an assumption of this thesis that all changes made to code are members of a set of particular types and that predictive measures can be built in to make these more abstract types of change easier to make. A prime example of a type of change is a change to data, a subtype of which is a change to data that doesn't affect the semantics of the data but changes its structure, such as the change depicted in Figure 3. Measures, in the form of adaptive code, can be built in to ensure that such changes are relatively easy to make because the evolution is localised and doesn't affect those other software entities that depend on the changing software entity.

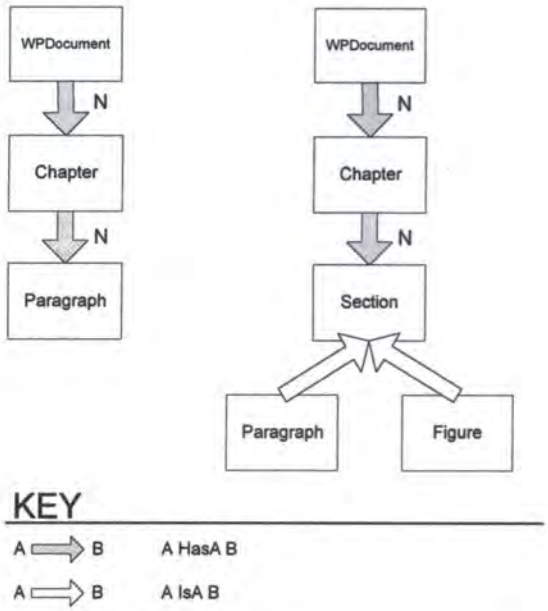


Figure 3 – Word Processor Document Evolution

As identified in chapter 2, software evolution results in a set of changes that are of two types:

- **Re-configuration type changes**: the existing software entities are able to satisfy the new requirement, with a suitable re-configuration which can be performed by changing parameters;
- **Adaptation/integration-type changes**: the existing software entities are unable to satisfy the new requirement and so new software entities are required, which must be integrated with the existing software entities. If the existing software entities are dependent on the new software entities, then the existing software entities may need adapting.

The types of changes that occur depend on the new requirement and the existing software entities in the software.

Re-configuration evolution is characterised by the fact that the new requirement can be expressed solely in terms of the existing software entities that constitute the software system. The change is a change in an instance or parameter. Adaptation/integration evolution, however, is characterised by the fact that the new requirement requires the integration of new software entities with the existing software entities that constitute the software system.

The dominant type of evolution is dependent on the type of system model chosen. For example, if the system model takes the form of a traditional low-level programming language, then most changes will require writing new code, and then integrating this new code with the existing code. Hence, most evolution is integrative. If, on the other hand, the system model takes the form of a domain-specific language, there is a lot of redundancy built into the language i.e. the language permits a wider range of functionality to be expressed than in a traditional programming language. As a result, many changes may be expressible as a re-configuration of the existing constructs in the language.

Software changes fall into six main categories as shown in Table 1. The change targets are of two types; instance or software entity, where an instance is an *InstanceOf* a software entity (see chapter 5 for description of the *InstanceOf* relationship). An instance change is re-configuration evolution. A software entity change is integration evolution. This is

related to intension and extension. Software entities typically only respond to changes in instances in their environment. Problems occur when new concepts, in the form of new software entities, occur in their environment. For example, a graph layout operation may be required to deal with 3D graphs when it has been implemented to deal only with 2D graphs.

Smith has categorised the changes that can conceivably occur in a software system as follows:

1. The change is within the basic capabilities of the system.
2. The change is within the basic capabilities of the system, with modifications to the ground rules that govern the behaviour of the system.
3. The change requires the addition of new capabilities.
4. The change doesn't conceptually fit in with the current system, neither in terms of the basic capabilities nor the ground rules that govern the behaviour of the system [Smith95a].

The basic problem rests in deciding whether the change can be accomplished within the basic capabilities of the software entities contained within the software system. 1 and 2 correspond to re-configuration evolution, 3 corresponds to integration evolution. In addition, Smith observes that requirements aren't necessarily compatible with a software system. For example, attempting to change a sort program to layout a graph would fall into category 4. Smith's classification is rather abstract and vague because it doesn't identify particular types of change within the context of each category. This thesis aims to identify types of change and their effects on other parts of software systems.

Change Target	Change Action	Example
Software Entity Instance	Add	A new server in a client-server architecture.
Software Entity Instance	Change/Adapt	Change an actual parameter[3] in order to choose a different sort implementation.
Software Entity Instance	Remove	Remove a service instance, or actual parameter.
Software Entity	Add	A new service.
Software Entity	Change/Adapt	Change in the behaviour of an existing service.
Software Entity	Remove	Remove a service or task.

Table 1 – Change Categories

Software should be able to determine the following after a change:

1. Which software entities are affected?

[3] An actual parameter is a formal parameter instance. Hence, the formal parameter is the software entity, and the actual parameter is a software entity instance.

2. How are these software entities affected?

Often there exists a two-way dependency between software entities. Imagine service x calls service y, then changes to y may affect x, and changes to x may affect y. For example, a user service calls a sort service to sort some data for it. The sort service could change in a number of ways:

- Change in interface i.e. type of data to be sorted;
- Change in behaviour (probably unlikely for a sort service, but the sort service could change with respect to constraints such as speed and space efficiency)

which may affect the user service. The user interface may also change:

- Sort data of a different type

which may affect the sort service if it isn't able to sort data of the different type.

The environment of a software entity is defined in chapter 5 section 2.1.2. Since it is the environment of the software entity that provides the impetus for the software entity to evolve/adapt, it is important from an evolution point of view to look at the link between the software entity and its environment.

There are two ways an FSE can interact with its environment:

- **Responsiveness** means that the FSE actively senses its environment through perception and adapts to changes within it. This is equivalent to polling;
- **Reactiveness** means that the FSE reacts to incoming messages and must adapt to changes in these messages, if this adaptation is within its basic capabilities.

Note that responsiveness implies "activeness" i.e. if a software entity is to be responsive to changes, then it must be an active software entity. A functional software entity's (FSE) environment is therefore "connected" to the FSE through responsiveness and reactiveness, both of which involve the transfer of messages. Therefore, the assumption is that the source of all evolution will be in the messages that services receive[4]. It is also assumed that all messages have a pre-defined structure or syntax (such as KQML [Finin93a]) which all services in the software system can parse. However, an FSE may not be able to interpret and therefore execute a message (this is discussed further in chapter 7 section 3.1.3).

[4] This view can be defended by the following argument. Computers are essentially "dumb", requiring outside help in order to make them do something useful in the form of code. Any evolution change in the software will be as a direct or indirect result of changes in the non-software environment (for example, human users, hardware sensors, the passage of time triggering a task etc). Therefore, any evolution change can percolate through the software (producing evolution changes in the software as they go) only through the messages which FSEs exchange. Any changes in the internal data structures of an FSE will, for example, originate in an outside influence.

Evolution then comes down to an FSE receiving a message that it can't directly interpret. This is essentially very late evolution, or dynamic evolution, in which evolution is driven directly by the software environment. Not all evolution is of this form, only evolution which follows the pattern of a client software entity requiring new capabilities of any server software entities which it uses.

As an example of this kind of environment-driven evolution, consider a graph FSE which receives a request to perform an "X" layout operation on a graph. The graph FSE doesn't know what an X layout operation is, so must evolve to be able to perform this functional capability. Since the software system itself is unable to write new functional capabilities, the software engineer must write them and integrate them with the existing FSEs in the software system. If, on the other hand, the output FSE receives a request to display graph nodes as squares instead of circles, and assuming that this FSE has the capability to draw squares, then re-configuration evolution will suffice. In this case, evolution of the data mappings between the two domains involved will satisfy new requirements.

A potential problem with integration is domain terminology/heterogeneity. Consider a client and server that need to communicate in order for the client to request the server to perform a service such as call-redirection, which consists of three main entities:

- Caller telephone number;
- First callee telephone number;
- Second callee telephone number.

Imagine that the first two telephone numbers are Telephone Company A numbers and the third telephone number is a telephone company B number. This means that the call-redirection service of telephone company A needs to inter-operate with the services of telephone company B. Specifically, the call redirect service of A should result in a request to the connect call service of B. There are two assumptions being made here;

- Messages are always sent to the "correct" FSE, which is able to interpret them;
- Both sender and receiver FSEs are using the same domain data semantics (or DEM + semantics).

Both of these are bad assumptions to make. It must instead be assumed that messages may indeed be sent to the wrong FSE and that FSEs may not be talking about the same thing, or may be talking about the same thing but using different models to do so. In order for an FSE to decide whether an incoming request is within its own capabilities, there must be some way for the FSE to identify when either incompatible terminology or domain models are being used. This reasoning could be based upon the DEM. However, there must be some way to be able to uniquely model a domain. The DEM, as a concept, on its own is not enough to accomplish this. Additional (semantic) models are required and need to be linked to the DEMs. The approach also assumes that the services will only need to be adapted in well-defined ways, as a result of changes in software entities that they depend on (for example, adaptation as a result of changes in the structure of the data that services use). Specifically, ad-hoc changes to services are not allowed because they are difficult to make and link to changes in requirements.

There are three major aspects to evolution in any software system:

- Data evolution;
- Functional Evolution;
- Control evolution.

The prevalence of these three types of evolution is dependent upon the type of application. Data-oriented applications (a simple example of which is a sort program utilising an input service, sort service and output service) will engage in more data evolution than control-oriented applications, an example of which is a simple telephone system consisting of a telephone switch and user handsets. The latter is more likely to engage in more control evolution than the former. However, the water is muddied somewhat by the fact that data evolution can trigger control evolution. For example, the seemingly simple addition of a new co-ordinate to a 2-D graph data model changes the semantics of graph layout and consequently the graph layout capability must be changed. This is not a change to the control aspect of the software but to the basic capabilities upon which the software is built.

This thesis focuses on product issues, as opposed to process issues, to improve software evolution. Modelling is recognised as the central problem when it comes to software evolution because the ease of making changes to satisfy a new requirement are dependent on the way the software system is currently modelled. It is also important in terms of what to model in order to aid evolution of the software. Having observed that software currently has no means to discover *how* it can evolve, and therefore that the onus is on the software engineer to understand the code and plan the changes necessary to meet the new requirements, this thesis proposes a set of software entities that can be used to model the real world and form the basis of evolution in software.

Hence, there are a number of main ideas on which this thesis is based:

- Increased semantic richness;
- Increased modularity;
- Reflection;
- Increased flexibility and adaptability of software entities to changes.

These are discussed in the following sections.

2.1 Increased Semantic Richness

Increased semantic richness advocates improving the semantic quality of software by allowing the software engineer to model more semantic aspects of software entities. It is related to increased modularity and the use of an increased number of software entities (or constructs) in the creation of software systems. It provides more targets, and thereby more interfaces, for evolution. The problem lies in determining concepts which:

- Are types within a domain (i.e. they are concepts which are stable within a domain, and are worth the effort of modelling), and;
- Provide a well-defined evolution space.

These concepts will typically be domain concepts which are implemented in terms of the basic abstractions in the software language being used, and may be difficult to determine. In addition, evolution may throw up new concepts of importance which are not explicitly modelled and have no evolution interface, and it may also remove existing concepts which are no longer important. These aspects of evolution are difficult to overcome.

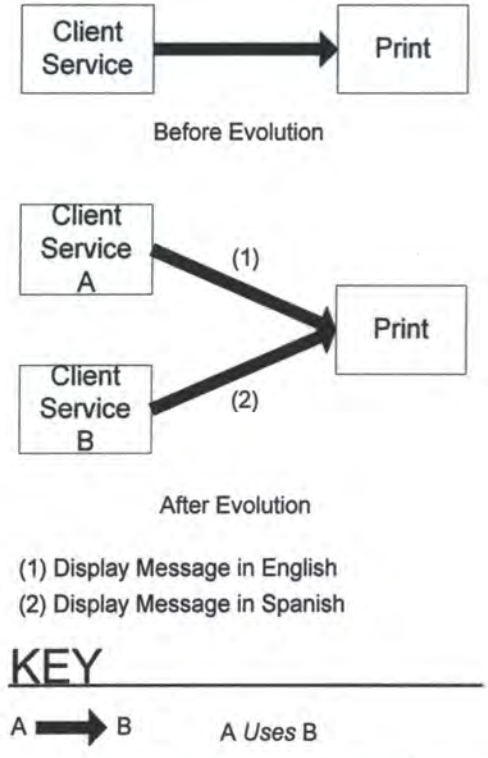


Figure 4 - Changing Client Requirements

As an example, consider the display of messages to the user in an arbitrary software system. Typically, this will be implemented by calling a function of the form Print (MessageText), which displays the message string "MessageText" on the screen. In this case, the display of messages is implicit and tightly coupled to the "Print" function. Next, a new requirement specifies that it must be possible to be able to display messages using a number of languages, such as English, Spanish and French. The existing implementation makes this difficult because of the lack of an API which provides an interface to the concept of "Display a message in a particular language on the screen"; this is shown in Figure 4 in which, after evolution, client service B's requirements result in the "Print" function being unable to satisfy the requirements of B any longer. Additionally, in this the "Print" function may be primitive i.e. not evolveable because it is part of a standard library. This means that evolution will require the creation of an API. Hence, the existence of an API for this concept before evolution occurs means that:

- The API provides a better interface to the concept than the existing implementation. In this example, replacing "Print (MessageText)" with "DisplayMessage (MessageText)" provides a better representation of the concept;

- It is easier to make changes that affect the implementation, but which won't affect clients of the abstraction as much. In this case, the concept of displaying messages as shown in Table 2.

Concept	Using Print	Using DisplayMessage
Display message in English	Print "Hello"	DisplayMessage ("Hello", "English")
Display message in French	Print "Bonjour"	DisplayMessage ("Hello", "French")
Display message in Spanish	Print "Hólà"	DisplayMessage ("Hello", "Spanish")

Table 2 - Concepts and Evolution

As another example, consider Figure 5 and a change that requires the creation and update times of all elements of a document to be recorded. The traditional way of doing this in an object-oriented model is to create a class that performs the recording of creation and update times through a particular interface, such as that shown in Figure 6, and have the relevant document elements specialise or use this class, with appropriate code that:

1. Links the constructor of the document element to the UpdateRecorder's creationTime () method;
2. Links the methods in the document element that alter the element's state to the UpdateRecorder's updateTime () method.

If the software is not designed in this way from the start, it can be difficult to integrate the desired functionality into the software. This is because the fact that the document elements are document elements is not explicitly encoded into the model, so that a change to the document elements as a whole is difficult to make. This is one form of lack of semantic richness.

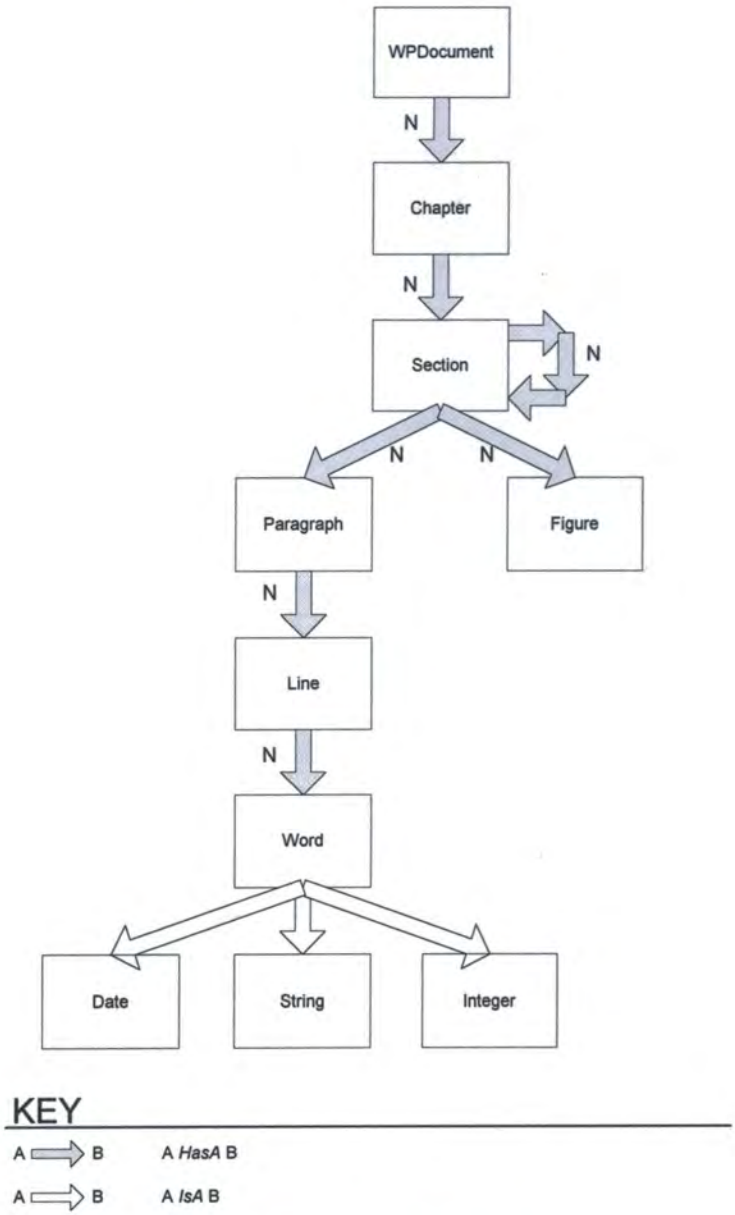


Figure 5 – Word Processing Document DEM

```
public Interface UpdateRecorder {
        public void creationTime (Time T);
        public void updateTime (Time T);
}
```

Figure 6 - Document Element Update Time Recorder

2.2 Increased Modularity (Separation of Concerns)

This thesis proposes an increase in the modularity of software. The idea is to increase the number of interfaces, which implies more indirection and a more broker-based architecture, which in turn provides the following advantages:

- An increase in the number of targets for evolution, increased localisation of evolution and more focused evolution;

- Providing the interfaces are designed appropriately, increased adaptability to change, because changes are hidden behind the interfaces.

The main disadvantage is less efficient code because extra levels of indirection is introduced. Chapter 5 describes a set of software entities, or modularisations, that achieve this end and provide a set of targets for evolution.

One of the advantages of increased modularity is that the resultant richer set of software entities means that one can use those software entities as potential targets for software evolution. Traditional software, on the other hand, has a less rich set of software entities and software evolution is therefore limited to more basic constructs, which is arguably more difficult to do. Maes, in her pioneering work on reflection, suggests reifying (representing at the meta level) as much as possible in the application domain [Maes87a]. This then allows one to perform meta operations on the reified object, which then provides a point of reference for the change to take place. Hence, the availability of more software entities allows the software evolution to be better mapped to the code.

There are two orthogonal aspects to software systems that must be captured within a modelling framework:

- Levels of abstraction (vertical layering);
- Structure within abstraction levels (horizontal layering).

Abstraction can be modelled using procedure calls, or inheritance in object-oriented software i.e. anywhere where a software entity needs to be *expressed in terms* of another software entity. This applies to both functionality and data. Structure within abstraction levels can be captured with the notion of software architecture that can be applied at any level of abstraction. As Garlan and Shaw point out, a software system may have different architectural styles at different levels of abstraction in the software [Garlan93a p4]. Abstraction is provided by the relationships:

- Data entity *HasA* Data Entity;
- Service *Uses* Service.

Software architecture/structure (for no change in abstraction) is provided by the relationships:

- Service *Uses* Message;
- Message *Calls* Service.

As Garlan and Shaw have recognised, every kind of software architecture has a number of things in common:

- A set of software entities;
- Software entities possess an interface, usually in the form of a set of messages which are mapped to local services – see [Bosch97a];
- There exists a set of semantically valid messages between each software entity.

Increased modularity has been espoused in many papers to varying degrees. For example, Gelernter and Carreiro separate out computation and co-ordination [Gelernter92a], which is very similar to Peterson et al. who separate out services (using objects) and behaviour (using controllers and executives) [Peterson94 p14]. Goldman et al also recognise the importance of modularity, in the form of separation of computation and communication (or co-ordination) [Goldman95a]. Lieberherr et al, in the Demeter system, modularise out the co-ordination or process elements of the software from the underlying class structure [Lopes94a]. In this way, certain changes to the underlying class structure don't affect the control components. These changes include generalisations and changes that don't conflict with existing requirements (and thereby don't change the assumptions that the control components make). Changes that will affect the control components, no matter how much adaptability is built-in to the control components, are those that include removing aspects of the class structure, for example. The structure-independent control knowledge is represented using a **propagation pattern** (see chapter 3). Ideally, the control (or co-ordination) knowledge which constitutes the behaviour of the software should be separated out from the functional capabilities of the software, so that changes in either will not affect the other. Smith shows how a distinction can be made between an object's internal description and external description: the internal description describes what the object *can* do, the external description describes what the object *will* do [Smith93a]. This subtle distinction has a number of consequences:

- The internal description of the object, which is state-based, provides an abstraction which exports a number of capabilities to the control aspects of the software system;
- The external description of the object links what the object can do with the rest of the software system by describing its interaction with other objects in the system. Hence, the object may be able to provide more capabilities than is required by the control aspects of the software system.

The inherently high coupling between control and functional capability results in difficulties for separation of concerns. Control knowledge is part of functional capabilities, and functional capabilities are part of control knowledge. This inherently tight coupling is a by-product of the engineered characteristic of software, in which the use of functional abstraction hides the details of the functional capabilities, and thereby encapsulates control knowledge too.

Gregor Kiczales and others at Xerox Parc, in their work on Aspect Oriented Programming (AOP) [Kiczales97a], also observe the benefits of separation of concerns by the identification of a set of cross-cutting aspects, which encapsulate some feature of the software (such as concurrency) which affects many parts of the software (hence its cross-cutting nature). Lalanda uses a different approach to achieve a similar end, by describing the intended behaviour abstractly in the form of a tuple (task, parameters, constraints), and allowing the software to choose the appropriate component to use [Lalanda97a p3]. The behaviour specification forms, in effect, an equivalence class of behaviours, for example, the behaviour specification:

(Sort, Data, Time < x milliseconds)

forms an equivalence class consisting of the sort algorithms that are able to satisfy the constraint that the time should be less than x milliseconds. This approach therefore implicitly de-couples the (abstract) control from the underlying software entity structure, as well as allowing a particular type of evolution to occur dynamically. Since the control plan

describes the intended behaviour only in abstract terms, the software is free to dynamically slot any component in that fulfils the behaviour description. For example, the following abstract control plan:

<Input, Sort, Output>

Control Element	Maximum Duration	Maximum Memory Use
Input	D_1	MU_1
Sort	D_2	MU_2
Output	D_3	MU_3

Table 3 - Sort Control Plan Constraints

describes a task that consists of performing input of data, sorting that data and then outputting that data. Table 3 expresses the constraints that must be met for each control element. Collectively, this set of constraints "...describes an equivalence class of desirable behaviours and in which currently enabled specific components may have graded relationships." [Lalanda97a p3].

However, Lalanda proposes the use of independent components i.e. components that are modelled independently - they're modelled only in terms of themselves, and not in terms of other components. This precludes defining valid combinations of components, unless a separate type of component is used for this purpose. It is also impractical because components need other components in order to carry out subtasks that they alone can't perform. Also, it may be the case that the component is highly context dependent and so must be expressed in terms of the data structures and components that already exist in the software system. An example of this is in the telephone switch domain, in which a new component to redirect calls needs to be integrated with the existing switch code. The redirect component is highly context dependent.

An important characteristic of the abstractions (or software entities) considered in this thesis is that they must be static and unchanging i.e. have a well-defined interface. For example, domain-independent, programming language constructs are abstractions that never change their form – loops are always loops, conditionals are always conditionals. For the domain-independent approach described in this thesis, the use of domain-independent abstractions is important. However, domain-independent abstractions will eventually be exhausted and one must turn to domain-dependent abstractions. But the static nature of these abstractions cannot be guaranteed, because such abstractions are typically only useable in a limited number of contexts.

Modularisation has always been a characteristic of modern software development methods because it helps create structured software and provides abstractions that help the software engineer tackle the complexity of software. This results in increased cohesion i.e. allowing related concepts to be grouped together. However, current modularity is ad-hoc and very dependent on the software engineer's modelling techniques and software development heuristics. Modularity can be controlled more by identifying types of modularity that can be performed, for example control in the form of processes, functional capability in the form of FSEs, data in the form of DEMs, data conversions, etc.

As Maes points out, increased modularity exposes interfaces to software entities that would not normally be explicitly modelled. This interface allows the software entity to be reflected upon and thereby manipulated [Maes87a]. In other words, increased modularity treats more of the software as first class objects. This is essentially the essence of reflection – the power it provides to manipulate aspects of software that are normally implicit. In traditional software, the only manipulate-able aspects of the software are:

- Procedures, which are manipulated through their interface - parameters;
- Classes, which are manipulated through their interface – methods.

However, there is a problem with increased modularity: the movement of complexity from components to interactions between components, so that one sacrifices increased complexity of interactions for simpler components. Of course, the complexity has to reside somewhere (see point below about conservation of complexity). In the case of increased modularity, the complexity moves from the un-modularised components to the relationships and interactions between the modularised components formed from the un-modularised components.

Conservation of complexity is an important aspect of modularity. A software system will always have a certain complexity, it's just a matter of where that complexity resides and is dependent on the modularisation chosen. Either components are complex and their interactions are fairly simple, or vice versa. It depends on what are chosen as the components. One can conceivably transform the software whilst keeping the functionality the same, so that particular components that were once complex are now made simpler, but the complexity has shifted elsewhere. Of course, there is also the issue of redundancy in data which can increase the complexity of the software for no gain in functionality. Data redundancy for efficiency purposes, for example, whereby a new data structure is created which contains the same data, but is accessed differently in order to allow this form of access to be performed more efficiently.

Note that the onus isn't on the software taking over the modelling of the real world, since humans are good at this, but on determining how software entities are related (for integration), how software entities can evolve, and how evolution in software entities affects other software entities. Procedural programming languages provide procedures and control constructs like conditionals, loops etc which are fairly low-level. Object-oriented approaches provide a richer set of constructs such as objects, classes, polymorphism etc. which are arguably still fairly low level. The idea of increased modularity recognises the fact that more abstract constructs are required.

The idea of increased modularity (or separation of concerns) espoused by Parnas in the 1970's has as one of its aims, the hiding of design decisions so that changes to these design decisions are hidden behind a well-defined interface. There are a number of problems with this approach:

- Changes to the encapsulated entity that fall outside the interface are not hidden;
- The approach requires some predictability on the part of the software engineer. The software engineer, when designing the software, needs to decide which abstractions to use that capture both the important aspects of what is being modelled and form a set of encapsulated entities that are prone to change. All changes can't be predicted however, and so changes will inevitably cut across the abstraction boundaries used;

- For various reasons (for example, laziness, lack of understanding of the abstraction required), the approach may not have been adopted across the application.

This thesis concludes that there is not enough semantic richness in the constructs that are currently used in the design of software systems. The choice of abstractions at design time don't match up with the set of abstractions that allow smooth and easy evolution of the software, because of the reasons described above. The solution of this thesis lies in the choice of constructs (software entities – see chapter 5) used in the design of software, which allow broader encapsulation of evolving entities and thereby improved evolution localisation. For example, changes that affect a specific set of FSEs should be translate-able into changes in a single software entity, rather than having to change every FSE affected. Of course, such a goal is not completely attainable, because it depends on being able to predict the groupings of FSEs that may change, and designing in the groupings at design time. However, it is reasonable to assume that particular groupings, such as the set of all FSEs or the set of all FSEs belonging to a particular domain, can be made that will allow the type of evolution identified above to be made more easily.

So, in summary, the abstractions or software entities used in this thesis (and described in chapter 5) have the following characteristics:

1. They are domain independent;
2. They are targets for evolution, and;
3. They can be reasoned about in isolation.

The first part is important because then the research can be applied across many domains. Using domain-specific software entities would result in domain-specific adaptability. 3 recognises the fact that a lot of code consists of aspects which can be extracted out and modelled as a separate software entity. For example, data conversions are the functional parts of code that change the representation of data used by a particular function to a representation required by the next function to be called. They are often implicit and hard-coded so that, when the data representation requirements of either function change, the data conversions are affected. The fact that such data conversions are hard-coded means that they may be difficult to find and the new conversion/transformation may be difficult to determine. Increased modularity results in treating data conversions as separate first-class software entities, with explicit dependencies between the functions on which they depend. Another example is the modularity of task (*what* is to be done) and *service* (how it is to be done).

This increased semantic richness of the modelling constructs is based on the identification of a number of types of evolution that would normally involve major changes in traditional software. For example, a separation of concerns between task (what to do) and service (how to do it, or implementation) means that a change in the implementation will only affect one part of the software (the mapping between task and service), as opposed to all points in which this mapping is implicitly used (as is the case with traditional software engineering techniques in which such a mapping construct is not explicitly used). This is also true of the data mapping construct, which provides a mapping between a source data structure and a target data structure (see chapter 6 section 3.4).

2.3 Increased Use of Indirection, Interfaces and Brokers

Many of the approaches to improving flexibility depend on an increased use of indirection [Kotula], such as the use of a broker architecture in which the brokers provide the indirection. For example, Schieffer uses object-oriented views in object-oriented databases to support evolution by essentially introducing a layer (a view) between the functions and the underlying data [Schieffer93a]. The view provides a level of indirection, or an interface, behind which certain changes to the underlying data can be hidden. The changes which are hidden are dependent on the design of the view, specifically the number of assumptions it makes about the underlying data. The more assumptions the view makes, the less it will shield the clients of the view from changes in the underlying data. Note that the introduction of indirection into an architecture or model implies the introduction of an interface.

2.4 Increased Evolveability

Evolveability is a software entity-directed term that refers to the ease of evolution of a software entity. A software entity can be anything from a data structure (for which evolveability means evolveability with respect to changes in data representation requirements of the user) to a whole software system (for which evolveability means evolveability with respect to changes in functional requirements of the user). Very few metrics exist that provide a quantitative, objective view of the evolveability of software. Keller provides an interesting model of the effort required to implement *any change* in a software system as:

> Total Effort (per change) = a[SLOC Changed] + b[#Modules Affected] + c[#Existing Functions Changed] + d[#Software Scenarios Changed] + e[Requirements Complexity] + f[Architectural Impact of Change]

where the constants (a, b, ..., f) can be determined from previous systems, subjective judgement or experimentation [Keller96a]. In contrast to existing maintenance cost models which focus on only specific properties of software systems such as function points [Bernstein95a], or lines of code, Keller recognises the influence of many factors on the calculation of maintenance effort, notably software architecture and requirements complexity. He also recognises that such a model is naturally evolving itself, so that effort is not a static measure or characteristic of a software system, but is dependent on the existing software as well as new requirements. However, the worth of the model has not been proven, and provides no way to include evolveability measures such as flexibility, adaptability and extensibility in the model. In addition, it ignores process issues such as the skills of the staff and the familiarity of staff with the domain (or domains) concerned. Keller's model provides a way to determine the effort of making a particular change, but not a way to determine the evolveability of a software system. Different changes will require different effort. Evolveability, however, is a more general characteristic of software systems that attempts to express the ease of making particular classes of change. For example, changing the way data structures are modelled in order to improve their evolveability with respect to new data requirements.

An important aspect of evolveability is the adaptability of software entities with respect to those other software entities on which they depend. Software flexibility is a system-wide term used to describe how easy it is to change a software

system as a whole in response to changes in requirements. A related term, software adaptability, refers to a software entity's ability to change in response to changes in entities on which they depend. This distinction is important because, although both are forms of adaptability, flexibility is generally regarded as a property of the software system as a whole, whereas adaptability is a term that refers to individual parts of a software system (the software entities).

2.4.1 Increased Adaptability, Flexibility and Extensibility

As pointed out in chapter 2 table 1, the difference between flexibility and adaptability (as used in this thesis) is that flexibility is concerned with the adaptability of software as a whole with respect to changes in requirements, whereas adaptability is used to describe how easy it is to change parts of software when changes are made to those parts of the software on which they depend.

The level of evolveability of a software entity is closely linked to the number of constraints placed on the software entity. These constraints take the form of design decisions and assumptions, which are often unavoidable because software must necessarily be constrained (there is no such thing as a completely un-constrained design). However, the more constraints that are imposed on a software entity, the less flexible and adaptable the software entity becomes. The road to improved evolveability involves limiting the constraints imposed on software entities, where possible. For example:

- As Hursch points out, the popularity of un-typed languages for prototyping stems from their flexibility in allowing changes to be made to data without adversely affecting the functional aspects of the software [Hursch95a p269];
- Object-oriented languages inherently impose restrictions on method visibility.

In general, increased adaptability is achieved through the limiting of assumptions. For example, a service software entity assumes a more general data structure so that when changes to the data occur, the service isn't affected.

Increasing the adaptability of software results in the creation of more interfaces, so that changes are more hidden by interfaces and hence the adaptability of clients of the interfaces is increased.

Previous approaches to achieving adaptability in software have resulted in "hard-coded" adaptability, or adaptability with respect to *known* concepts. Networking is a good example of adaptability, in which networking software adapts to changes in the environment (consisting of switches, satellite links, optical-fibre links, buffers etc.) in order to provide Quality of Service [Tanenbaum96a] to the user. This form of adaptability is provided in terms of an algorithm which adapts the behaviour of the software to known variations in the environment. These variations occur to known concepts in the environment and the responses are typically hard-coded in.

Crelier describes an approach to a specific form of adaptability, which focuses on the adaptability of client functions with respect to *extensions* in the interfaces of other functions which they use [Crelier95a]. His research developed a reflection-based framework that attaches meta-information on interfaces for each module in a software system. This meta-information is used at link-time to

The framework is limited to interface extensions (addition of interface objects), and so doesn't deal with interface adaptations that involve:

- Removal of interface objects;
- Changing of interface objects.

However, as Crelier points out, these types of changes will probably invalidate clients of the interface, whereas extensions to the interface shouldn't invalidate a client because the client doesn't see the extensions. Of course, this is dependent on whether or not the interface extensions lie within the behaviour of the server. If the interface extension occurs as a side-effect of a change in the behaviour of the server, then clients of the server will inevitably have to use the interface extensions. In other words, the extension doesn't invalidate the requirements the server satisfies but occurs as a result of a reconfiguration change. However, the client needs to be re-expressed in terms of the new configuration.

An extensible software model is characterised by its ability to integrate changes into an existing design by extending or enhancing existing aspects of the design. Object-oriented models, for example, are extensible because they provide an interface for extensions (the class abstraction). The extensibility of a model is, however, hindered by new requirements which conflict with existing requirements (because changes will probably require the removal of existing concepts) and by requirements which are not expressible in terms of existing concepts.

Extensibility is concerned with the ability of a software language, model or architecture to support extensions to a design. Support for extensibility can be from a number of measures:

- Hooks;
- The "right" abstractions, so that extensions can subclass them;
- Lack of conflicts between new and existing requirements.

The major disadvantage of extensibility is that it's limited to extensions, as suggested by its name, and an extension doesn't permit all types of integration evolution. It is limited to integration evolution in which an *existing* abstraction is being changed.

An important aspect of extensibility is **dynamic extensibility**, which is concerned with how easy it is to add new software entities to a running software system without affecting the software system. A good example of a software model which permits this type of extensibility (though in a limited domain) is that of blackboard architectures.

2.4.2 Types of Adaptability

The existence of many types of entities and many inter-dependencies between these entities results in many different types of adaptability. Probably the most well-known type of adaptability is that which exists between a function and the data structures it uses [Lopes94a]. This is not the only type however and, although adaptability is constrained to that of software entities with respect to changes in other entities (i.e. the impacts of changes are one-way and in the direction of

entities to software entities), with appropriate modelling adaptability of software entities with respect to changes in, for example requirements, can be approached. A list of examples of adaptability is shown in Table 4, where the adaptability is expressed in terms of adaptability of a client software entity with respect to changes in a server entity.

Client	Server	Reference	Comments
Functional Software Entity	Data[5]	Chapter 7	
Functional Software Entity	Software Architecture	Chapter 7	
Functional Software Entity	Functional Software Entity	Chapter 7	
Functional Software Entity	Requirements or Desires	Chapter 7	
Data Entity Model	Data Modelling Requirements	Chapter 6	Adaptation of the data structure in order to satisfy new data modelling requirements.
Data Mapping	Data Entity Model	Chapter 6	
Data Entity Model	Data Entity Model	Chapter 6	For example, adaptability of a class with respect to changes in its superclass(es)
Data Instance Model	Data Entity Model	Chapter 6	Adaptability of data instances with respect to the data model of which they are an instance.

Table 4 - Types of Adaptability

Most approaches to adaptable software have been of the following types:

- Adaptability of object model/schema with respect to class model/schema, pursued mainly by the database community [Banerjee87a];
- Adaptability of methods with respect to data, such as the work of researchers at Northeastern University [Lieberherr93a, Lopes94a]. Hursch briefly discusses the concept of method evolution, which is the adaptation of methods with respect to changes in class structure in order to maintain behavioural consistency[6] [Hursch95a p36].

Other types of adaptability have been largely ignored, a state of affairs which this thesis aims to rectify.

2.5 Localisation of Evolution

Localisation is an important concept for evolution and maintenance since it localises changes to smaller areas of the code than normal. Localisation of evolution solves the same types of problems as increased adaptability using a different

[5] Such as class structure or DEM.

[6] Behavioural consistency is attained by adapting the functional aspects of the software with respect to the programming language and data schema/model in order ensure that aspects of the original behaviour still satisfy the new requirements.

approach. Whereas increased adaptability focuses on design for change at the client software entity, localisation of evolution focuses on design for change at the server software entity by limiting the effects of evolution to as small a set of software entities as possible. For example, the use of special data-access services (see chapter 6 section 5) limit the number of affected services when data changes. Another example is the use of DIM paths (see chapter 6 section 3.3) which are similar to data-access services in that they separate out the data item(s) to access and how to access those services. Hence, the increased use of interfaces (to separate out what to do from how to do it) allows software evolution to be as localised as possible.

2.6 Increased Generality

The rule is to choose the most general representation possible for all software entities. There are three aspects to this, stemming from the fact that there are three distinct types of software entity in a software system:

- Data components (DEMs);
- Functional components (FSEs, tasks);
- Structure, or software architecture.

As an example, consider relational database schemas as an example of inflexible data structures. Suppose we have the relational table:

Employers : (Employer-Name, Skills-Required, WWW, Address)

(where Skills-Required, WWW and Address are all dependent only on the primary key, Employer-Name) and we want each employer to be able to have a number of departments, such that Skills-Required is dependent on the particular department. Then, tables Employers' and Departments are as follows:

Employers' : (Employer-Name, Address, WWW)
Departments : (Employer-Name, Department, Skills-Required)

Note that Skills-Required is dependent on both Employer-Name and Department, which are both the primary key, whilst Address is dependent only on Employer-Name. The simple addition of a new attribute has caused problems for the relational model, resulting in the need to re-design both the tables.

The problem is that the most general representation for data is a hierarchy or tree. Even though data may start off as a non-tree data structure, such as a relational model or a list, the data may eventually change to a graph. This will inevitably require the data structure that is changing to be re-designed. The solution to overcoming inflexibility, therefore, is by the use of the most general data representation possible, the graph.

As an example, consider the hard-coded dependencies between the functional elements of a software system. Smith suggests a shift away from reactive software and towards adaptive software [Smith95a p3],. The former has rigid

interfaces which changes in the environment may break. The latter implies a different way of writing software, based upon identifying the set of services that "are necessary for tasks within a certain domain, and identifying the strategies, or 'ground-rules' to be used by agents for negotiation and co-operation, rather than designing the details of the interaction itself." This is a form of interaction generalisation, whereby the interactions between the functional components in a software system are generalised, rather than having designed-in, hard-coded interaction mechanisms which are liable to change. This is related to software architecture generalisation, a method of generalising the interface between FSEs and software architecture, in order to ease ripple effects when the software architecture changes.

2.7 Change Prediction

Change prediction is the prediction of changes to software entities at an appropriate level of abstraction, by the inclusion of evolution operators that allow the software entity to be evolved by providing an evolution interface. In software where particular changes can be predicted, specific measures may be built in to allow those types of changes to be made more easily, at least locally. For example, in a graphics editor that allows the user to edit different shapes, a typical change is the ability to edit a new type of shape. Using knowledge gleaned from other systems and from domain knowledge, software engineers can predict the types of change that may occur in the software system and then build measures into the software to make these types of changes easier than they normally would be. This could be achieved by making the change possible through a change in a parameter value, arguably the easiest type of change. It's important to recognise, however, that ripple effects may overwhelm the ease of change of parameter value-type changes. Being able to convert a higher level change to a parameter value change would considerably ease the implementation of that change, even though it requires the software engineer to build this capability into the code. In management terms, this may be undesirable, but it can be viewed as shifting some of the inevitable costs of software evoluton to the software development stage, thereby taking advantage of project members' knowledge of the software, knowledge that is so often not available after post-deployment evolution and maintenance. However, this approach will only work for particular changes i.e. those that have been predicted through experience with similar systems.

A major tenet of this work is that changes, in general, can't be predicted. However, another major tenet of this work is that classes of change can be predicted, and these classes of change are intimately tied to the types of abstractions chosen to model the real world. These abstractions can evolve in particular ways and, together, provide the basis for evolution of the whole software. The usefulness of this observation depends heavily on the level of abstraction of the classes of change identified, and whether these classes are domain-independent or not. For example, if the class is the class of *all* data structures then it is fairly obvious that such a class will involve some evolution at some point in the future. This class is very abstract and provides no help. A more concrete class of changes is those changes that apply to a particular data structure entity or a specific software architecture entity.

One form of change type taxonomy is along abstraction boundaries. Changes range from fairly concrete, domain-specific changes such as changes to variable values, to abstract domain-independent changes such as changes to the software architecture. As Hursch points out, the target software entities of the change depend on the level of abstraction:

> "At the lowest level of the system, normal system operations perform extensional changes, whereas software evolution performs intensional changes. In other words, extensional changes do what the system is designed to do while intensional changes modify the design of the system." [Hursch95a p22]

2.8 Modelling

Modelling is an important aspect of this research because it can't be avoided when writing software. This is true whether the software is small or large and whether the software is developed using a traditional low-level language approach, or whether transformations are used, or whether a domain-oriented language is used in the implementation. This is because any piece of software is a model of the real world and as such has to change when the real world changes or when an error in the model has been discovered [Lehman98a]. Modelling is also important because it provides a way of analysing and reasoning about the software with a view to changing it. This task is normally performed by a (group of) software engineer(s), but is increasingly being assigned to the software itself, mainly in situations where the changes are very simple and don't require large changes to the model. The extent of the modelling inherent in software can be viewed along a spectrum from software which has everything modelled explicitly, to software in which there is mostly implicit modelling i.e. concepts are hard-coded in and difficult to reason about/manipulate. Thus, there are two aspects to modelling when applied to software. The most familiar one is the traditional modelling that occurs when the software engineer is encoding aspects of the real world in terms of a software language. The other aspect is modelling that involves the software engineer producing models of the earlier models. These types of models are termed reflective (or self) models because they model parts of the software system itself. Hence, the first type of model models the real world, the second type of model models the real world models.

The ease with which evolution occurs in a model is dependent on the model chosen. It is important to include in the model anything that may change within the software. This is the same as saying that everything that may change in the future is parameterised so that the change can be effected by a simple change to the value of a parameter (arguably the simplest type of change in a software system). Take the Windows 95 task bar as a simple example. Imagine that Windows 95 had been written with the task bar along the bottom of the screen, and imagine a user wanting to change this so that the task bar could also be positioned vertically down the left-hand side of the screen. If the position of the task bar was included in the model (as a parameter in the code for example), then this change could be made quite easily. However, this contradicts one of the assumptions of this work that, in general, changes cannot be predicted. So, the question is how can everything that may need to be changed later on be included in the model so that it can be changed by the simple alteration of a parameter?

Simple changes could possibly be predicted, such as the change in position of a task bar. Typically, the positioning of a task bar is implicit in the code i.e. implicit in a call to a graphics API function. Changing this requires a stage of program comprehension to understand what needs to be changed, followed by the change being made, followed by the determination and management of ripple effects. Making this positioning explicit, in the form of a higher level model, should make changing it easier.

It has traditionally been very difficult to evolve software based on changes to non-functional requirements such as speed and efficiency. This is because there is no clear path from the change in requirements to the actual changes that need to be made to the code. Meta-level architectures, particularly the open implementation approach [Kiczales93b], aim to change this by allowing the implementation for a particular abstraction to be altered at a higher level than the code. This is exemplified by the sort abstraction that defines a particular behaviour parameterised by efficiency or speed concerns, which determine the particular sort algorithm used. Hence, for the same behaviour, the algorithm is parameterised by speed concerns.

2.9 The Evolution Process

As pointed out in chapter 1, this research can be classified as an enabling technology, concerned with product issues of evolution rather than process issues. The process-independence of the approach means that any evolution process model could feasibly take advantage of the work described in this thesis. However, in order to show the potential advantages of the work described herein, it is assumed that some form of change request software evolution model is being used [Bennett91a]. It is assumed that, since this work is concerned with improving the product, the software entities affected by the change request have already been identified, an assumption that is backed up by the existence of such information in a typical change request form [Sommerville92a]. In this case, an improvement in evolveability will inevitably help in making the required changes to the software. The increased modularity of the software may also improve the ease of mapping particular kinds of changes, such as changes in data conversions, or changes in task knowledge, to the software itself.

3 Software Entity Adaptation and Integration

The adaptation of a software entity can be viewed equivalently as the integration of an aspect of that software entity with another software entity. Hence, adaptation and integration are essentially two sides of the same coin. For example, adaptation of a service is equivalent to integration of a message with the body of that service. Hence, refinement or adaptation of a software entity is equivalent to adding and removing software entities at a lower level of abstraction. For example, refinement of a software architecture is equivalent to adding and removing components and connectors, within the constraints of the software architecture.

All software systems and software entities possess an intension and an extension. Extensional changes occur during the normal operation of the software system, whereas intensional changes modify the design of the software system. In terms of a software system's entity and instance models, an extensional change is a change that occurs *within* the configuration of the entity model, and to the configuration of the instance model i.e. the instance model is re-configured but the entity model isn't. Therefore, extensional changes occur as a result of either normal system operation or re-configuration evolution, where new instances of existing software entities are made and these new instances are supported by the existing software entity model. For example, imagine that a user wants to draw a window in a different colour, and that there exists a concrete service to draw windows called drawWindow (DIM Window) and the DEM shown in Figure 7, where colours are represented by red, green and blue components in the range 0-255. The change can be performed by altering a parameter to the drawWindow service. This is a re-configuration change because the software

system "knows" about the new colour: the change doesn't require any changes to the entity model (including changes to services).

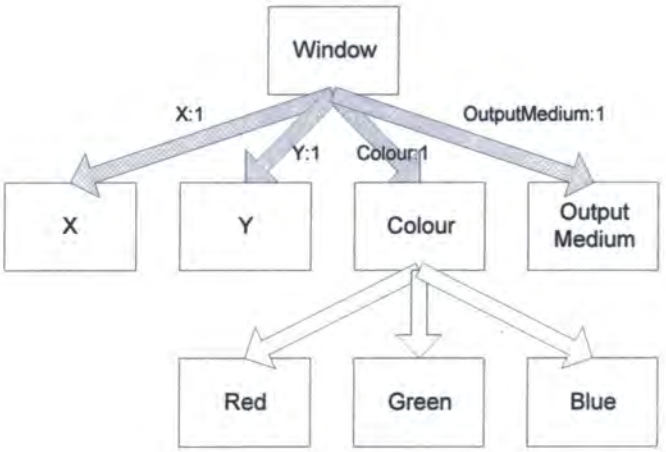


Figure 7 - DEM_{Window}

It is the primitive software entities in the software entity model that provide the extensional information in software. That is, the primitive entities describe how the software actually is, not how it might be. This latter task is the realm of the intension of the software. So, given these definitions, a re-configuration of the software means using the *existing primitive software entities* to provide a new extension for the software; the primitive entities are in effect re-configured in order to provide the new requirement. This assumes that there is redundancy built into the primitive software entities i.e. that a re-configuration is possible. If not, then the only solution is the use of an evolution space, which requires shifting the level of abstraction up one or more levels. For example, if the evolution can't be carried out within the existing software architecture extension, one must move along the *IsA* a relationship to the next higher software architecture abstraction.

A software system consists of dependencies (or uses relationships) between the engineered software entities. Each dependency is a potential source of a ripple effect, and consists of:

- A client software entity;
- A server software entity.

The client software entity uses the server software entity, which is the cause of the dependency. Changes in the server software entity are potential causes of ripple effects in the client software entity. There are two approaches to overcoming this:

- Make the client more adaptable to changes in the server;
- Localise changes to the server;

When both of these fail, ripple effects will inevitably occur.

The evolution spaces of the software entities:

- Messages;
- Tasks

provide a parameterised interface to a set of different implementations. In the case of messages, their parameters (which include the type of service call e.g. local procedure call, RPC etc., whether the service call is synchronous or asynchronous etc.) determine which type of message is chosen. The parameters provide a space of possibilities e.g. synchronous RPC, asynchronous local call etc. In the case of services, the actual parameters to the task determines the concrete service to call (if there is one). An heuristic is that as many assumptions made in the service are extracted out as possible, with the assumption that not all assumptions will be known. Hence, in the case of both services and messages, assumptions about whether the service call is synchronous/asynchronous, and whether it is local or remote are deferred until run-time, allowing a choice to be made about the desired implementation. The identification of as many assumptions as possible increases the flexibility of the software without huge effort.

Every reified software entity, by definition of being reified, possesses a manipulation interface, which provides a well-defined interface for adapting or making changes to the software entity. This manipulation interface can be fairly un-restrictive (allowing unconstrained changes), or may be constrained by semantic considerations. An example of the latter is DIMs, by virtue of their high dependency on a specific DEM which means that the DIM must be an instance of the DEM. Chapters 6 and 7 provide, for each software entity identified in chapter 5, a set of manipulations which constitute an evolution space for the software entity.

There are evolution spaces:

- Where the target entity model configuration is unknown;
- Where the target entity is known:
  - e.g. bubble sort → quick-sort;
  - e.g. compiler filter architecture → compiler shared repository architecture;
  - e.g. 2Dgraph DEM → Drawing DEM.

  In this case, evolution consists of a mapping from the source entity to the target entity. For example, mapping a compiler filter architecture (the source entity model) to a compiler shared repository architecture (the target entity model) involves the use of architecture evolution operators to refine existing sub-entities in the source entity model.

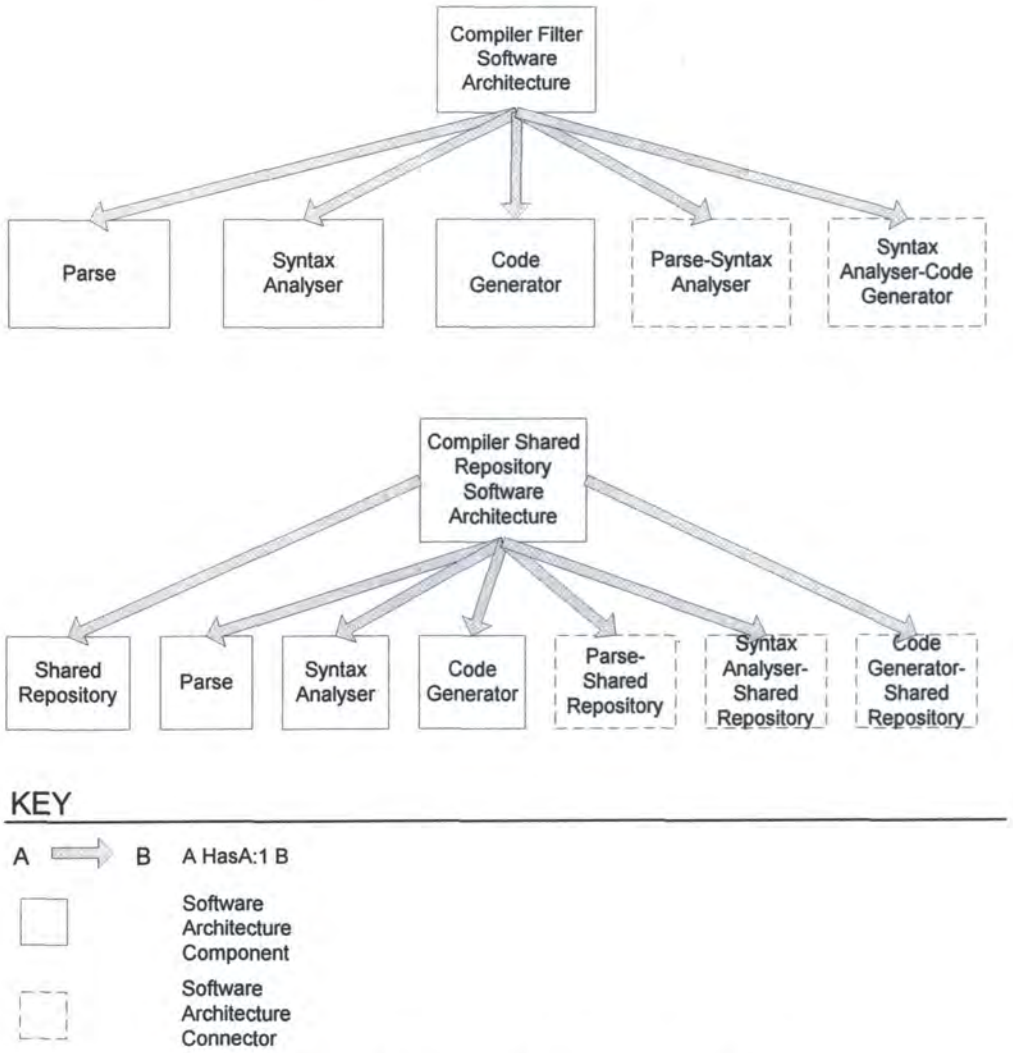


Figure 8 - Software Architecture to Software Architecture Mapping

For example, in Figure 8, a mapping from a filter compiler architecture to a shared repository compiler architecture has the following characteristics:

- The software architecture components are mapped as-is. This depends on how the components are modelled and, specifically, how de-coupled they are from the specific architecture that is being used. Any use of architecture-specific information in the components will inevitably mean that the components will be affected by changes in the software architecture;
- The software architecture connectors, which contain the architecture-specific information, need to be transformed as they are mapped, which is dependant on the mapping.

The thesis returns to this subject in chapter 7, when it considers the adaptability of FSEs with respect to software architecture changes.

3.1 Software Entity Evolution Spaces (Types of Change)

Evolution spaces are a way for the software engineer to describe how a particular software entity can evolve. They are provided for each of the software entities introduced in chapter 5. Software entities that refine these software entities may need to refine the evolution space operators provided as standard. Component adaptation is quite complex because the ways in which a component (in the form of a class, with state and operations/methods) can change are quite numerous. The state can be augmented with new state members, existing state members can be adapted (for example, through subclassing), new methods can be added, existing ones re-defined etc.

For example, a filter software entity's evolution space can describe the limits/constraints on evolution that existing architectures simply don't allow one to do. Interfaces in Java [Flanagan97a], for example, only provide constraints in the form of a set of required methods. This is not enough to be able to describe more complex constraints. For example, consider a filter component. In order to describe the evolution constraints for this, one needs to describe the *types* of services that the component consists of. Hence, constraints consist of the following types:

- Constraints on *behaviour* of service;
- Constraints on data – type of data used → use of a DEM.

A good example of the evolution space approach in practice (although it isn't actually called this) is visual programming in software like Visual Basic, Borland Delphi and Borland Visual C++. These tools impose a structure on developing software systems that consists first of designing the user interface and then proceeding to write the code that implements that interface. An event-based software architecture is used to implement the link between the user interface code and the application code. The important point to note for this discussion is that the design of the user interface follows a pattern that basically consists of choosing existing user interface components (such as dialog boxes, buttons, menus), laying them out on a form and refining their characteristics to the user's needs. For example, a menu component may be refined by choosing whether it is a pull-down or pop-up menu. Such refinements allow the user to customise their user interface in pre-defined ways, much like the evolution operators are used to refine other software entities in this thesis.

An evolution space is defined by:

- A set of software entities, the things that evolve;
- A set of evolution operators/transformation rules, which specify how software entities can evolve,and;
- A set of constraints, which cut down the evolution space by preventing certain (operation, entity) combinations from occurring which are semantic. Semantic constraints prevent certain sequences of services that don't make semantic sense e.g. in a sort control element, the typical sequence is <input, sort, output>. It doesn't make much sense to have <sort, input, output>.

In another related area of research, Ward and others have developed Wide Spectrum Languages (WSLs) [Bull95a] that use a similar idea, that of using a set of transformation rules on an initial set of entities. In WSL, however, the initial

entities are the software system and the transformation rules are semantic-preserving operators that encapsulate both how to transform a given software entity and the semantic-preserving constraints on doing this.

Cazzola et al describe an evolution space approach for software architecure [Cazzola97a]. Their transformation rules are separated into a similar set of distinct operator types:

- Add new entity;
- Remove existing entity;

(with the notable exception of a change/rename operator) and a set of domain-specific constraints that are not necessarily semantic-preserving [Cazzola97a]. The constraints describe (either using a model as in this work or using a language of some sort) the allowed mappings between software entities. This can be accomplished by encapsulating the constraints in the form of a graph-based model that shows the relationships between software entities and thereby describes how "legal" transformations can occur by adhering to the model. For example, a task "sort" can be transformed into "Quicksort" or "Bubblesort" using an "implements" mapping. So, given a start configuration of software entities, every configuration reachable from this start configuration can be determined by repeated application of the operators to the start configuration, ensuring that the constraints are met at each application.

Hursch uses a similar approach to evolve object-oriented software [Hursch95a]. In his work, the limits of software evolution are defined by the cross product of:

- A set of software entities:
  - classes;
  - method signatures;
  - attributes of classes;
  - access control;
  - inheritance relationships;
  - composition/part-of relationships, and;
- A set of evolution operators [Hursch95a p54].

The software entities in his research are object-oriented software entities. The evolution operators which can be applied to the software entities are add, remove and change/rename. The separation of evolution operators into these three types seems an intuitive classification. However, in this case the change/rename operator is not really required because of the recursive nature of evolution spaces: a software entity can be adapted by adding, removing or adapting its constituent software entities. Hence, we have the set of evolution space operators defined in Table 5.

Evolution space Operator Name	Description
Add Software Entity	Adapt a software entity by adding a software entity sub-part to it.
Remove Software Entity	Adapt a software entity by removing a software entity sub-part from it.

Table 5 - Evolution space Operators

In order to utilise the evolution space approach, it is important to choose an appropriate set of entities, operators and constraints. In their case study, Cazzola et al choose the entities, operators and constraints shown in Table 6.

Entity	Operators	Model	Constraints
Tracks	Insert and Delete	Graph (Tracks as Nodes, Relationships as Track connections)	No more/no less than 2 parents for each track entity i.e. forms a ring

Table 6 - Cazzola et als' Software Architecture Evolution space

However, add and remove operators based on an architectural type model aren't the only operators that can be used, even though they are the only domain independent ones. Other operators are domain dependent, based on some form of change operator that converts a particular entity into something else by changing some of its characteristics. The relationship between the old entity and the new entity can take many forms, such as:

- Implements;
- Specialises;
- Same behaviour, different implementation.

For example, a bubble sort service entity in a control element can be changed to a quick sort entity by use of a change of the latter variety, where the behaviour stays the same but certain other characteristics change. In this example, non-functional characteristics such as speed and efficiency change. The constraints are generally domain specific and dependent on the particular application.

The evolution space approach utilises constraint models when determining the software configurations "reachable" from the current software configuration. These models are graphs with directed relationships between the entities (or concepts) contained within them. An evolution space walks through the model only in the direction of these relationships, so cannot explore any parts of the model "above" the concepts used in the current configuration. This is desirable because it doesn't allow the control trace to be generalised (by moving back up the model) and make it invalid with respect to the requirements. e.g. generalising a sort to a filter is generally not what is wanted. For example, the sort process in Figure 9 will allow more of the model to be explored than the sort process in Figure 10 because it uses on average more higher level concepts.

Evolution space theory assumes that evolution occurs with respect to a context, so that within that context the evolution proceeds along a well-defined path determined by the evolution operators within that context. For example, software

architecture styles provide a grammar (consisting of components and connectors) that allows one to describe many software architectures. The redundancy built into the software architecture style, along with operators that allow one to add and remove components and connectors, and constraints that limit the types of evolution, provides a context within which a particular architecture can evolve.

<Input.*, Sort.*, Output.*>

Figure 9 - Abstract Sort Process[7]

<Input.DialogInput, Sort.BubbleSort, Output.*>

Figure 10 - More Concrete Sort Process

Programming language records also provide a context for evolution of their members. The evolution of the record as a whole is determined by the evolution of its members, whose evolution is determined by the types of the members. The type provides a context for evolution of variables. If the evolution has been forseen by the software (i.e. the change can be expressed in terms of instantiation of an existing part of the model) then the evolution is really an extensional change.

For example, consider a client-server system with three types of component:

- Clients;
- Listener Servers, which listen for task requests from clients;
- Task Servers, which perform tasks for clients, at the request of listener servers.

This is a common occurrence in the UNIX operating system, for example, where daemon processes (listener servers) dynamically spawn another process (task server) to carry out the task while they go back to their job of listening for task requests. A potential change to the software system is to add a new task server. This is an extensional change because it involves the instantiation of an existing modelled entity. However, the addition of, for example, a completely new type of server is an intensional change, or evolution, because the existing software system doesn't know how to use the new change;

Bosch identifies three types of (component) adaptation:

- Copy-paste;
- Inheritance;
- Wrapping [Bosch97a].

along with another type, superimposition, which allows adaptation to be viewed as the composition of an adaptation type with an object. Hence, Bosch takes a different approach to adaptation than most of the other current approaches, by modularising the adaptation process components into:

---

[7] The notation <...> represents a sequence of message sends (or calls). Each element of the sequence is the target service part of the message; an asterisk "*" is a wildcard meant to represent many services with similar behaviour (so "Input.*" represents "any input service").

- A set of software entities to be adapted;
- A set of adaptation entity types.

thereby providing a taxonomy of component adaptation techniques, organised into three categories as shown in Table 7.

Component Adaptation Category	Component Adaptation Type	Comments
Changes to Component Interface	Change method names	Changes the method names so that a client can use the new interface.
	Interface restriction	Restriction to particular parts of the interface on a per-client basis.
	State restriction	Restriction to parts of the interface that use only particular subset of the component state, on a per-client basis.
Component Composition	Delegation of requests	A component passes on a request to another component.
	Aggregation in encapsulating component	An encapsulating component accepts requests for a number of components, delegating requests to them.
	Acquaintance selection	Each component has a set of components (acquaitance set) to which they can send requests. Acquaintance selection means choosing an appropriate acquaintance to which to delegate a request.
Component Monitoring	Implicit invocation	A particular type of change in the component triggers notification of other components.
	Observer notification	A specialisation of the implicit invocation adaptation type e.g. MVC in smalltalk, in which components express an interest in changes in other components.
	State monitoring	Notification when state exceeds particular boundary conditions.

Table 7 - Bosch's Component Adaptation Taxonomy

The interface is the set of messages for which there exists a mapping to a method in the component. There is a one-to-one mapping from messages to methods. Component monitoring is a form of integration e.g. observer notification superimposes observeable behaviour on an existing component – the observation behaviour is integrated with the existing behaviour of the component. Bosch's set of adaptation types can all be expressed in terms of component interface changes, rather than component body changes.

The problem with Bosch's approach is that the applicability and reusability of adaptation types may be fairly limited, because adaptations are typically fairly context dependent. Context independent adaptations exist only at higher levels of abstraction, where common patterns of adaptation can be identified. At lower levels of abstraction, adaptations are typically heavily tied-in with individual requirements, so the adaptation will work only for that particular combination of requirement and existing software.

An evolution space is defined by a set of operators and constraints that define how a software entity can evolve. The basic operators are of three types:

- Add a component of a software entity;
- Remove a component of a software entity;
- Change a component of a software entity.

The meaning of these operators depends upon the entity to which they are being applied, as Table 8 shows.

Software Entity	Add Operator	Remove Operator	Refine Operator	Reference (Chapter)
Process	Add message	Remove existing message	Refine existing message	7
Task	Add formal parameter	Remove existing formal parameter	Refine existing formal parameter	7
Service	Add message	Remove existing message	Refine existing message	7
Message	Add actual parameter	Remove existing actual parameter	Refine existing actual parameter	7
DEM	Add data entity	Remove existing data entity	Refine existing data entity	6
DEM Mapping	Add Mapping	Remove existing mapping	Refine existing Mapping	6
Formal Parameter	See DEM entry.			
Architecture	Add component and/or connector	Remove existing component and/or connector	Change existing component and/or connector	7

Table 8 - Software Entity Evolution Types

Each software entity's evolution space is limited by constraints, which are typically requirements. The evolution space of a DIM, for example, is constrained by the fact that the DIM must always be consistent with respect to the DEM of which it is an instance, which is governed by the *InstanceOf* relationship discussed in chapter 5 section 2.1.5.

Software Instance	Add Operator	Remove Operator	Refine Operator	Reference (Chapter)
Process Instance	Add message	Remove existing message	Refine existing message	7
Task Instance	Add formal parameter	Remove existing formal parameter	Refine existing formal parameter	7
Service Instance	Add message	Remove existing message	Refine existing message	7
Message Instance	Add actual parameter	Remove existing actual parameter	Refine existing actual parameter	7
DIM	Add data instance	Remove existing data instance	Refine existing data instance	6
Actual Parameter	See DIM entry.			

Table 9 - Software Instance Evolution Types

Chapters 6 and 7 describe the evolution spaces of the software entities and instances described in chapter 5.

4 Analysing Software Entity Adaptability and Ripple Effect Types

The approach is based on recognising that evolution of a software entity can be of a number of different types, each of which may have mutually exclusive characteristics which determine the effect on any dependent software entities. For example, determining how a data structure can evolve is based on applying the idea of an evolution space to the data structure, which depends on the particular model chosen for the data structure. If, for the sake of argument, one chooses the DEM as the data model (DEMs are discussed in detail in chapter 6) and applies the evolution space operators add and remove, then the evolution types consist of those discussed in chapter 6 section 4.1 (which are listed in Figure 11 for convenience).

Data Adaptation Type
Add a New DEM
Remove an Existing DEM
Add a New Data Entity
Remove an Existing Data Entity
Add a New *HasA* Relationship
Remove an Existing *HasA* Relationship
Add a New *IsA* Relationship
Remove an Existing *IsA* Relationship

Figure 11 - DEM Evolution/Adaptation Types

These evolution types can be successfully linked to particular types of ripple effect or adaptability in dependent software entities. However, if refined further, they can provide more detailed information on adaptability and ripple effect types.

For example, as discussed further in chapter 6, the addition of a new data entity to a DEM can be characterised as either a domain-changing change or not. If the change is known to alter the domain (because of the breaking of constraints built into the model which "document" certain assumptions about stability of the domain, for example, that a particular data entity doesn't change its semantics), then this will have particular effects on, for example, the services of the domain. An enlightening example is that of the 2D graph domain. Adding a new co-ordinate to the node entity transforms the domain as a whole into the 3D graph domain, and invalidates any graph layout services which use the domain data. Hence, the recognition of particular characteristics of software entity changes, and their relation to effects on any dependent software entities, can greatly aid the software evolution process.

5 Summary, Discussion and Conclusion

The main disadvantage of the evolution framework presented in this thesis is the increased use of modelling that has to take place. Initial work and expense is sacrificed for later cost and effort savings. The software should be more evolveable. It is hopefully more evolveable in the sense that extensions to the model are easy to perform. Developing the meta-models consisting of software entities will probably be more time-consuming than current languages, models and architectures (although there is no empirical evidence to support this view) but will provide a richer set of abstractions for the software engineer, in addition to a way to trace ripple effects and increase the evolveability of software. Additionally, software is more self-documented, an increasingly common characteristic of software, as evidenced by increasing use of reflection and other self-modelling approaches, such as work by Karakostas on teleological maintenance, the linking of concepts used in the analysis and design stages of software development with those used in the implementation in order to trace dependencies [Karakostas90a]. Time and effort that would normally be spent on developing documentation for the software can be spent on creating "internal documentation" for the software in the form of reflective models, giving the software a reflective capability. In this way, the documentation is more closely linked to the actual software artefacts that it represents.

The choice of software entities is an important aspect of the work describe herein. Hursch's choice of software entities is based on an object-oriented architecture or model [Hursch95a], and is classified into three categories:

- Software schema (class) model;
- Object model, and;
- Method/behaviour model

His work is concerned with the preservation of consistency of the three models with respect to each other, as a result of changes to the schema model.

An important point to make is that the costs (in terms of money and effort) of initial investment in software development, specifically explicit self-modelling, is sacrificed for later increased evolution efficiency. This is not so bad when one considers that documentation is always separate from the software it documents and it is difficult to link elements of the documentation with aspects of the software. Movement of this documentation to the software environment, and emphasising these links is a central part of this work [Karakostas90a].

In summary, the approach described in this thesis is:

1. The identification of a set of software entities, that are prone to change and possess improved evolveability with respect to the software entities in existing software architectures, models and languages;
2. An analysis of the ways in which the software entities identified in 1 can change, and the effects of these changes on dependent software entities;
3. An analysis of how evolveable the software entities identified in 1 can be made;
4. If software entities can't be made completely evolveable (because certain assumptions inherently persist), which ripple effects occur and how can they be overcome?

Chapter 5

A Set of Highly Evolveable Software Entities

1 Introduction

There are a number of aspects to software evolution, as discussed in chapter 2. There are also a number of still-unsolved problems, as chapter 2 concludes. Chapter 3 discusses a set of approaches to dealing with the problems identified in chapter 2. Of these problems, this thesis is concerned with the evolveability of software. Existing approaches to this particular problem have been based on ripple-effect/impact analysis, which attempts to determine *what* aspects of software will be affected by a change to the software. Recent research on Adaptive Software (AP) [Lieberherr93a, Liebererr94a, Lopes94a] has attempted to approach the problem from a different perspective, by improving the adaptability of aspects of software when changes occur to other aspects of the software on which they depend. However, the work focuses only on the adaptability of behaviour when changes occur to the underlying class structure on which it depends. This thesis identifies a set of software entities, aspects of code that:

- Can be collectively used to model the real world. These are identified and described in this chapter. They are similar to constructs in existing software languages, models and architectures, with the additional characteristic of improved evolveability with respect to existing models;
- Are dependent on each other in different ways (see section 2.1);
- Encapsulate assumptions about those software entities on which they are dependent;
- Are potential targets of evolution.

The approach described in this thesis is based on:

- Increasing the inherent flexibility of software by limiting assumptions and increasing the number of abstractions. The general rule is: separation of concerns leads to increased use of interfaces and abstractions which leads to inherent evolveability (flexibility and adaptability), because changes are hidden behind interfaces. Changes outside the chosen interfaces will, of course, inevitably result in ripple effects but this is, in general, reduced. Increased separation of concerns also creates more potential targets of evolution, and thereby inherently improves the locality of changes. In addition, adaptability is improved by making the interfaces more generic;
- Increasing the inherent flexibility with respect to new requirements by lessening the constraints imposed by the software architecture. The most notable example of this kind of flexibility is the flexibility with which functions can call other functions without being constrained to a set of particular functions. OO models, in contrast, limit which methods can be called by the inherent constraints imposed by the class structure;
- Making explicit as many dependencies between software entities as possible, so that ripple effects can be determined and traced i.e. the effects of a change can be determined, allowing the software engineer to determine what needs to change and what doesn't need to change as a result of evolution. This is not the case with traditional

software architectures, models and languages, in which dependencies are encapsulated in interfaces which fail to capture all the dependencies that exist. For example, non-functional dependencies are not explicitly modelled in interfaces. Neither is behavioural information.

Another aim of the thesis is to make individual software entities *more* adaptable with respect to changes in other software entities on which they are dependent, in the same way that the propagation patterns of Demeter are adaptive with respect to certain changes in the class structure. Note that the software entities described in this chapter are not completely adaptable i.e. they are not adaptable with respect to all changes. Where complete adaptability is not possible, the following are performed:

- The determination of *how* the software entities change – the evolution spaces that document, for each software entity, a set of evolution types;
- The determination of *how* adaptation types affect other software entities which are dependent on the software entity that is changing, if at all.

The way in which this increased adaptability is achieved is through the limiting of assumptions made by software entities about their **environment**[1].

Increased parameterisation is also emphasised, as it is in open implementations (see chapter 3 section 8). For example, use of a particular message-passing style can be chosen by parameterisation, so that changes can be effected through re-configuration evolution i.e. changes to parameters. The parameter provides an evolution space for the aspect of the code that it parameterises; so a message type parameter provides an evolution space that consists of all the types of messages (RPC, local, CORBA, shared memory etc) that are allowed. Often, a particular concept, such as messages, can have different implementations. These different implementations can be parameterised, like in open implementations and work on reflection. However, each implementation will make assumptions that may be difficult to change when the implementation is changed, because the alternative implementation will have conflicting assumptions. For example, synchronous procedure calls and asynchronous procedure calls have conflicting assumptions about *when* any result is returned. Changing from a local procedure call to a remote procedure call requires undoing any assumptions made. This is difficult in traditional software languages which assume local procedure calls by default, because the interface for local procedure calls (built in to the language) is different than for remote procedure calls (which may be provided by a library). The solution is to move to common, more generic interfaces in which conflicting assumptions are merged e.g. asynchronous procedure call interface subsumes synchronous procedure call interface, so always use an asynchronous procedure call interface.

Of course, appropriate mappings are required in order to change from, say, local message calls to RPC. This idea is akin to the CLOS meta-object protocol where meta classes provide a set of implementations of the aspects of CLOS that they reify [Kiczales91a]. For example, a class meta-class reifies classes, and provides a mechanism for altering the behaviour

[1] A software entity's environment consists of those software entities that directly or indirectly use or are used by the software entity concerned (see section 2.1.2).

of classes such as inheritance behaviour, by altering the meta-object of a particular class. In effect, the software engineer chooses from an evolution space of possibilities in the meta-class in order to change the behaviour of the class.

The limits of evolution of a software entity are encapsulated in the evolution space of that software entity. The evolution of a software entity may affect any dependants of the software entity. It is therefore important to determine dependencies and make them explicit, so that a lack of adaptability means that the resultant ripple effect types can be determined, and the appropriate evolution of those affected software entities performed.

The aim of this chapter is simply to introduce the software entities and describe them along with their inter-relationships. In addition, the evolution spaces of these software entities is also described along with their reflective attributes, which comprise the interface which a software entity exports to its clients. A major feature of these software entities in general is their abstractness. The reason for this is mainly as a consequence of trying to keep the software entities domain independent and thereby improve the applicability of the model in many domains and contexts. The main aim of the software entity model (described in section 2) is as a harness for the "documentation" of reflective information about the software entities, reflective information which is used during software evolution. The intention is not to discuss the flexibility and adaptability of software entities, which is described in chapters 6, 7 and 8.

2 Software Entity Models

Software entity models provide a way to model the software entities in a software system and their inter-relationships. A software entity model is a directed graph that consists of the software entities described in section 3 connected by the relationships described in section 2.1. This section describes the overall structure of a software entity model and the relationships that exist between software entities.

A software entity model consists of two parts:

- Domain-independent software entities. These form a static/unchanging set of software entities used by the domain-specific software entities. For example, the service software entity;
- Domain-specific software entities. These are expressed in terms of both domain independent software entities and other domain-specific software entities, through any relationship except *InstanceOf*. For example, a sort service software entity is a sort domain software entity, which is a specialisation (*IsA*) of the service software entity.

Both domain-independent and domain-specific software entities can be of two types:

- Concrete software entities, which reside at the leaves of the software entity model, and cannot be specialised. For example, algorithms, or primitive data entities such as integers and characters, or standard software architectures such as an event-based architecture;
- Abstract software entities, which reside at non-leaf nodes in the software entity model.

The domain-independent software entities are described in section 3. The domain-specific software entities model the application domain and are therefore domain-specific.

There exists an orthogonal classification for software entities:

- Primitive software entities don't evolve. This may be because the software entity is an algorithm, or because it is a data structure which doesn't change because it models data in a well-defined, static domain such as the 2D graph domain;
- Non-primitive software entities, which are prone to evolution.

2.1 Software Entity Relationships

2.1.1 Dependencies Between Software Entities

Each software entity is not a stand-alone entity, but part of a larger model. Thus, there exist dependencies between all of the software entities identified in section 3. These dependencies can potentially result in ripple effects in software systems, depending on the adaptability of the software entities. For example, changing an existing data conversion may result in changes to the services which use the data conversion, which in turn may result in the creation of new services. Note that this is not just restricted to changes in the software entity's explicit interface, because changes in other aspects of the software entity can potentially cause ripple effects. For example, the behaviour of a service is not traditionally considered part of a service's interface, even though changes in it may invalidate clients and cause ripple effects. Changing a class model, for example, can invalidate any methods that depend on the pre-evolution structure of the class model before evolution (this will include all methods that refer to classes outside their class i.e. any methods that "walk" the class structure). It is important to determine these dependencies in order to determine the (ripple) effects of evolution in a software entity on other software entities in the software.

Ripple effects are regarded as a problem in software evolution because they must be dealt with in order to "re-stabilise" the software system; that is, bring the behaviour of the software to a state that is valid with respect to the set of requirements that includes the existing requirements encapsulated in the system and new requirements that triggered the evolution in the first place. The term "ripple effect" as used in this thesis is defined in Table 1 along with two other definitions of importance.

Term	Definition
Primary Evolution	Primary evolution is evolution which occurs directly as a result of new requirements.
Secondary Evolution	Secondary evolution occurs as a result of new requirements breaking existing requirements and assumptions encapsulated in the software, which may cause ripple effects.
Software Entity Environment	A software entity's environment is defined as those (software or non-software) entities that either affect or are affected by the software entity (see Figure 1). The operator Env (Entity) returns the set of entities in the environment of "Entity".

Ripple Effect	A Ripple Effect is a transformation of a software entity that is within the evolution space of that software entity and which contributes to bringing the software system as a whole back to a consistent configuration of entities and instances with respect to the original requirements. A consistent configuration is a configuration in which the software entities and instances satisfy the original requirements. So, for example, a change to the class structure in an OO software system will often invalidate some or all of the methods (depending on the type of change), requiring them to be transformed in some well-defined way so that their behaviour is the same as before the evolution of the class structure. Note that this transformation is often performed by the software engineer in an ad hoc manner, even though, as Hursch describes in [Hursch95a], there exist standard transformations for particular kinds of changes to class structure.

Table 1 - Definitions

A dependency is a directed relationship between two software entities which may signify that evolution of the child software entity can potentially result in evolution of the parent software entity, depending on the adaptability of the parent and the type of evolution of the child. It is important to be able to model the relationships between different software entities that make up a software system, in order to be able to determine how changing a particular software entity will affect other software entities. So, for example, a change to data structure may produce changes to the functional software entities which use the data structure. In addition, it is also important to determine all potential dependencies in a software system. This chapter attempts to go some way in accomplishing this.

Hence, a distinction is made between those changes in a child which don't affect any parents because the parent is adaptable, and changes which do affect the parents. This allows a software engineer to identify when a potential trigger for evolution, caused by a change in a child, turns into a real trigger for evolution because the change is outside the extended interface provided by adaptability.

In software entity figures, directed lines depict the relationships between the software entities (which are depicted by boxes) and are drawn such that the software entity at the head of the arrow is the parent of the relationship and the software entity at the tail of the arrow is the child of the relationship. For example, the relationship between "Task" and "Service" is read as "Service *Implements* Task".

There is also an aggregation (1:N) relationship between software entities such that a non-primitive software entity can structurally contain one or more software entities, which are its children. For example, a software system may contain many subsystem software entities, which may themselves contain many subsystem software entities. A primitive software entity doesn't structurally contain any other software entities (which is implied by the "primitive" in its title).

2.1.2 The Environment of a Software Entity

Each software entity possesses an environment which consists of those entities, hardware or software, that affect or are affected by the software entity in question. The terms "affects" and "affected-by" are intentionally vague and abstract.

They indicate that the relationships between software entities can take on many forms, and are dependent on the software entities.

A particularly special environment is the system entity environment that consists of at least one non-software entity. This environment is typically known as the system boundary and consists of entities over which the software may have no control, unless it has effectors that can directly or indirectly act on this environment. Direct actions include controlling hardware actions whilst indirect actions are typical of Lehman's E-type applications in which the software itself is a part of the system being modelled [Lehman85b]. An example of this type of action is the software changing working practices.

Ideally, changes will be driven by the environment. The trigger can either be direct (such as a service receiving a message that can't be handled by its interface) or indirect (such as a change in the environment of a software entity invalidating one of that entity's assumptions). In practice, every software entity's environment can't be made complete. For example, the environment of the system software entity (the software entity that encapsulates the whole software system) is very complicated because it consists of entities in the real world, which is itself complicated. Such factors as users' desires and beliefs can affect the system software entity and trigger changes, but this sort of information is difficult to measure. For example, the trigger for a change such as adding an "onhold" feature to a telephone switch originates in the environment not modelled by the software, so there is no trigger inherent in the software. To track down the trigger in this case, one must turn to the wider environment i.e. that encapsulated in the non-software entities such as:

- Hardware;
- Software owners;
- Software users, etc.

In this case, the trigger would be a change in the desires of the software owners to introduce an onhold feature. This is problematic for traditional software that doesn't model such parts of the environment. It is also problematic in the sense that it may be difficult to determine what needs to be modelled in order to allow all evolution triggers to be included in the software. The corollary to this is determining the boundaries of what has to be modelled. As Smith comments, "In order to create an artificial egg, you need to create the whole universe" i.e. it helps to understand an entity's environment in order to understand the entity itself [Smith95a].

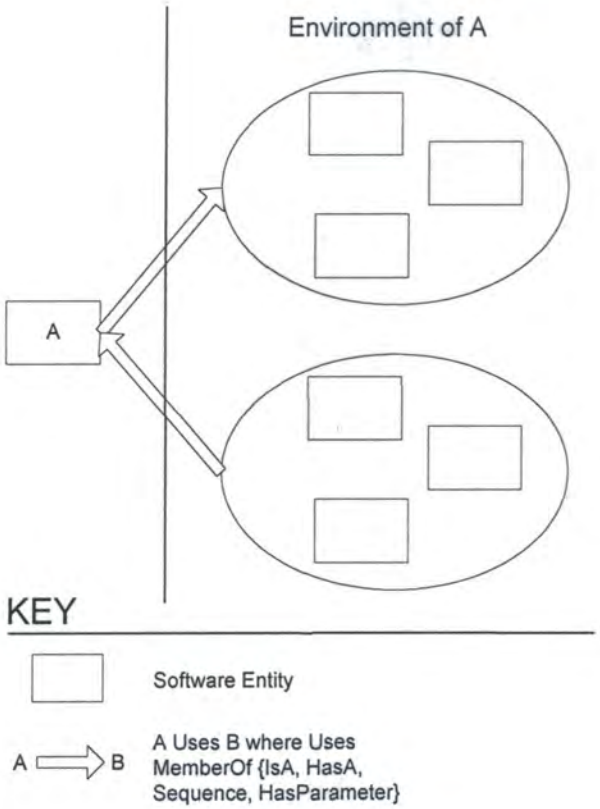


Figure 1 – The Environment of a Software Entity

Software entities can take on the role of either clients or servers, or both. Clients *Use* servers. Servers are *UsedBy* clients. Clients are also termed "dependants", because they depend on particular aspects of the servers they use, such as the interface of that server, or the behaviour of the server (if it is an FSE). This is an extension of the terminology used by Meek in [Meek95a], which he uses to distinguish between the participants in a procedure call. A client's environment consists of those servers to which it is related. A server's environment consists of those clients to which it is related. Table 2 shows the set of possible relationships that can exist between software entities, and the software entities between which these software entities exist.

	Relationship	Parent-Child	Comment
1	*Implements*	Service-Task Service Instance-Task Instance	See section 2.1.3.
	ImplementedBy	Task-Service Task Instance-Service Instance	
2	*IsA*	DEM-DEM Task-Task	See section 2.1.4.
	Generalises	DEM-DEM Task-Task	

3	*InstanceOf*	DIM-DEM Message Instance-Message Service Instance-Service Task Instance-Task	See section 2.1.5.
	HasInstance	DEM-DIM Message-Message Instance Service-Service Instance Task-Task Instance	
4	*HasA*:<Cardinality>	Entity-Entity	See section 2.1.6.
	PartOf:<Cardinality>	Entity-Entity	
5	*HasA*	Entity Instance-Entity Instance	See section 2.1.7.
	Part of	Entity Instance-Entity Instance	
6	*Uses*	Service Instance-DIM	See section 2.1.8.
	UsedBy	DIM-Service Instance	
7	*Calls*	Service-Message Service Instance-Message Instance	See section 2.1.9.
	CalledBy	Message-Service Message Instance-Service Instance	
8	*Produces*	Message-DIM	See section 3.1.2.1.
	ProducedBy	DIM-Message	
9	*Removes*	Message-DIM	See section 3.1.2.1.
	RemovedBy	DIM-Message	
10	*Updates*	Message-DIM	See section 3.1.2.1.
	UpdatedBy	DIM-Message	

Table 2 - Software Entity Model Relationships

There are certain rules or constraints that must be met for a software entity model to be valid/consistent:

Every non-primitive software entity must be the parent of an "<entity> *IsA* <entity>" relationship i.e. it must specialise an existing entity in the model. This implicitly assumes that all future software entities (at any level of abstraction) can be incorporated into the model, which is probably not achievable. However, the assumption is that most future software entities will be expressible in terms of the software entity model *at some level of abstraction* even if the existing software entity (the entity already in the software entity model) must be refined to the new entity.

Term	Definition
Software entity sub-model	The entity model reachable from an entity through child relationships.
Instance sub-model	The instance model reachable from an instance through child relationships.

Structure sub-model	The entity sub-model containing only *HasA* relationships.
Software entity's structure	The structure sub-model of the entity.

Table 3 - Definitions

2.1.3 The *Implements* Relationship

This relationship exists between a service or service instance parent and a task or task instance child, and allows the software engineer to express functional abstraction with respect to behaviour. This means that certain aspects of the behaviour of the parent service are left parameterised, much as functional abstraction with respect to data allows the software engineer to design functions which accept different data through data parameters. The parameters being described here provide another kind of interface which allows the caller of the service to vary the behaviour of the service by choosing appropriate behavioural parameters, with the constraint that the resulting behaviour will always satisfy the child task. So, for example, consider the partial software entity model shown in Figure 2. The task "Sort" provides a functional abstraction which allows both its behaviour and data to be refined using the task interface. The behavioural refinement is selected by the behavioural parameters. In this example, the service "BubbleSort" implements the "Sort" task. The implements relationship is similar to the *IsA* relationship for data entities because both allow the software engineer to express refinement of software entities. The distinction between them is in how the refinement proceeds, which is different because the notion of refinement for data is different than for function.

There can be many implementations of a task, as shown in Figure 3. Given a requirement "layout graph G" expressed in terms of a task, which is quite abstract, algorithms (or services) A_1 to A_n implement it i.e. any one of these could be used. The one to be used is then typically dependent upon non-functional requirements, design constraints or user preferences.

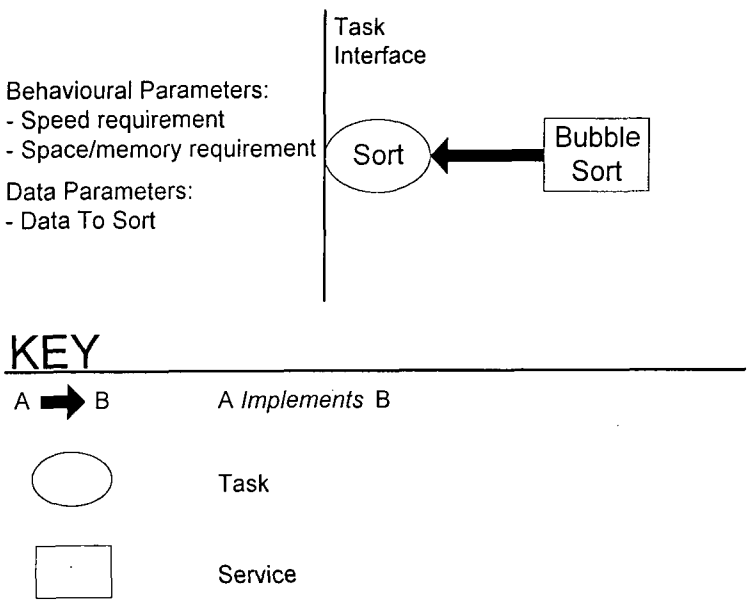


Figure 2 – The Implements Relationship

This relationship is similar to the open implementation approach, described in chapter 3, in which software entities possess two interfaces:

- A requirements interface, which defines *what* is to be done, in the form of a task. For example, "layout graph";
- An implementation interface, which defines *how* the requirement is actually implemented, allowing a choice between a number of different implementations based on non-functional requirements, as shown in Figure 3. The service software entity abstraction provides this capability.

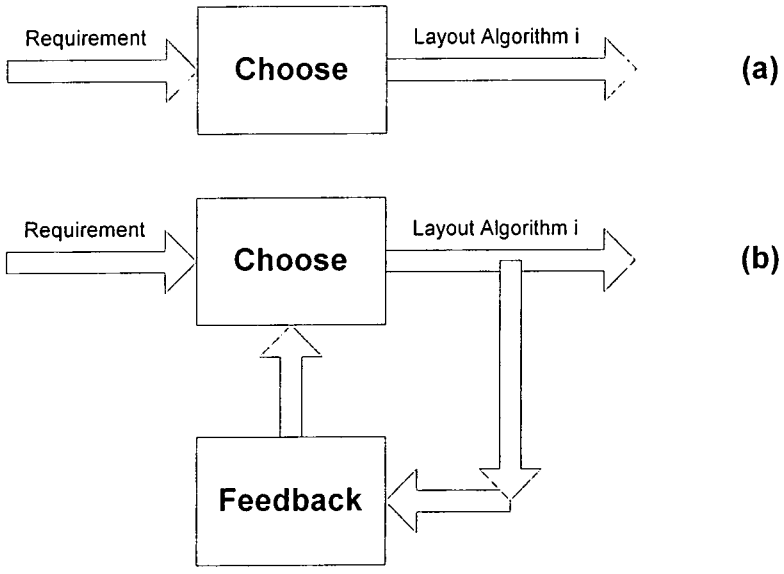


Figure 3 - Different Implementations of a Requirement

This can be expressed in terms of two sets of parameters; behavioural parameters and data parameters, as shown in Figure 2.

2.1.4 The *IsA* Relationship

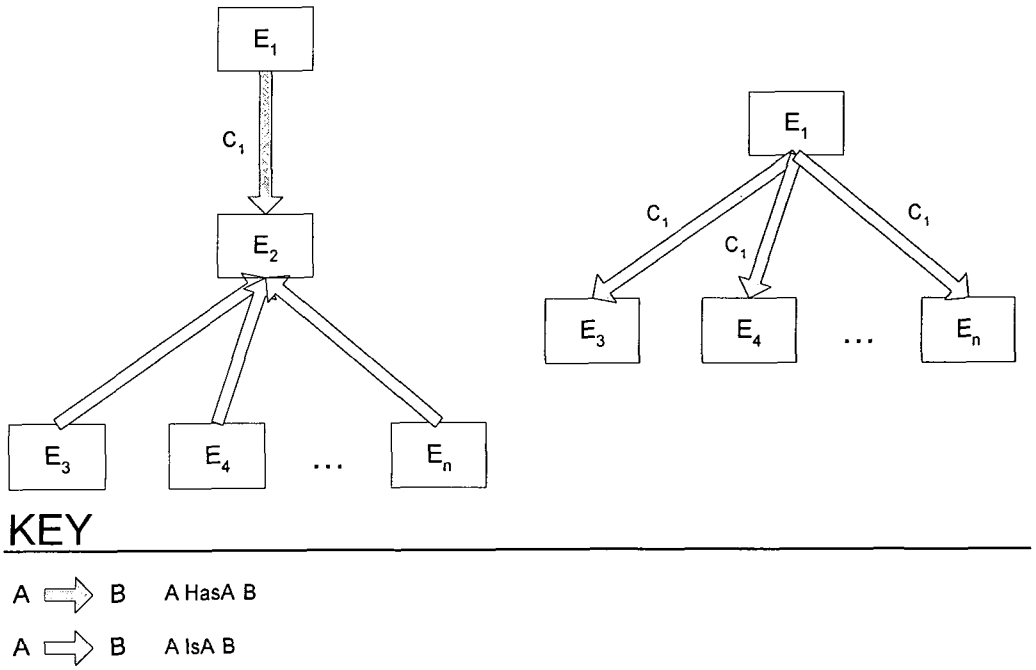


Figure 4 - Modelling the *IsA* Relationship

The *IsA* relationship is a relationship which exists between two data entities and has object-oriented inheritance semantics, such that the data entity parent inherits from the data entity child. *IsA* relationships are sometimes implicit as depicted in Figure 4, which shows the equivalence between *IsA* and *HasA* relationships.

If a data entity is a parent in more than one *IsA* relationship, then the *IsA* relationships are effectively or'ed. In this case, there must be a common interface to each child data entity, so that the interface provided by the "connector" data entity is not broken by different interfaces to the two types of style. In this case, the interface is the common interface afforded by the procedure and parameters abstraction. For example, the connector style can either be RPC (Remote Procedure Call) or shared memory.

DEMs permit only single inheritance. However, this shouldn't preclude the modelling of multiple inheritance, as Templ discusses in [Templ93a]. His approach is to represent multiple inheritance using single inheritance, and his conclusion is that multiple inheritance is "...purely syntactic sugar for expressing situations where objects have a one to one relationship and refer to each other." which doesn't increase the expressive power of the model.

2.1.5 The *InstanceOf* Relationship

An *InstanceOf* relationship exists between:

- Data instances and data entities;
- Service instances and services;
- Message instances and messages;

- Task instances and tasks.

The instance sub-model must be a valid instance of the structure sub-model of the software entity being instantiated. Each child instance must be an instance of a child software entity of the software entity being instantiated.

The rules which an *InstanceOf* relationship imposes on the parent and child ensure consistency of the relationship and represent basic *InstanceOf* semantics which are present in all software languages. There are three cases to consider in the case of DIMs and DEMs:

- **DIM *InstanceOf* DEM:** Base Data Instance *InstanceOf* Base Data Entity;
- **Data Instance *InstanceOf* Data Entity:** ∀ DIM Path ∈ submodel (Data Instance), ∃ DEM Path • DIM Path *InstanceOf* DEM Path;
- **DIM Path *InstanceOf* DEM Path:** there is a direct mapping between each data instance in the DIM Path and each data entity in the DEM Path, and each relationship in the DIM Path is valid with respect to the corresponding relationship in the DEM Path.

2.1.6 The *HasA*:<Cardinality> Relationship

The *HasA* relationship describes an aggregation relationship between two data entities of cardinality C, such that there can be at most C child instances of the parent instance. The valid cardinalities are:

- =x, where x is a natural number. There must be exactly x child entities;
- <=x, where x is a natural number. There can be <=x child entities;
- N. There can be any number of child entities.

In some cases, there is a need to be able to express relationships between different cardinalities. Take the spreadsheet DEM shown in Figure 5 as an example. The number of rows should be equal to the number of cells in a column, and the number of columns should be equal to the number of cells in a row. Hence, the cardinalities in the figure for Spreadsheet.Row.Cells and Spreadsheet.Column.Cells are given in terms of the cardinalities of Spreadsheet.Columns and Spreadsheet.Rows, respectively.

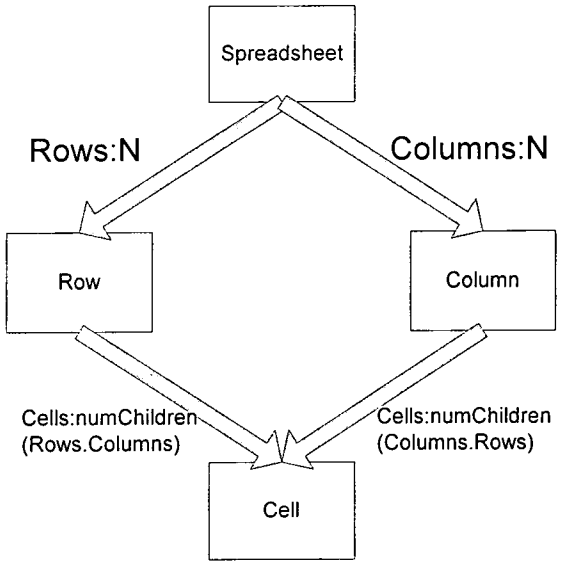


Figure 5 - Spreadsheet DEM and Cardinality Relationships

2.1.7 The *HasA*^{I} Relationship

The *HasA*^{I} relationship describes an aggregation relationship between two data instances and must be consistent with respect to the DEM of which the DIM that contains the two data instances is an instance. The rules for this consistency are:

- Every *HasA*^{I} relationship in the DIM must be consistent with respect to the relationships in the DEM. This means that the parent and child instances in the *HasA*^{I} relationship must be valid with respect to a *HasA* relationship in the DEM;
- The number of child instances must not exceed the cardinality expressed in the DEM.

2.1.8 The *Uses* Relationship

The primary use for this relationship is to specify a particular service instances' actual parameters, in which case the first form of the relationship is used.

2.1.9 The *Calls* Relationship

The *Calls* relationship depicts a function delegation relationship between two services by relating the calling service to a message which encapsulates the service call. Figure 13 shows how a service calls another service through a message, or how a service instance calls another service instance through a message instance.

2.2 The Removal of a Software Entity

The removal of a software entity has implications on the software entity model from which it has been removed. If E_y is the software entity to be removed then, for every pair $({}_xR^i_y, {}_yR^j_z)$ where ${}_xR^i_y \in$ parent relationships of E_y and ${}_xR^j_y \in$child relationships of E_y, map $({}_xR^i_y, {}_yR^j_z) \rightarrow {}_xR^k_z$ where (R^i, R^j) maps to R^k. A software entity may have a number of parent

relationships and child relationships, which will be left dangling when the software entity is removed. The mapping (R_i, R_j) → R_k describes how pairs of dangling relationships are combined into single relationships, as shown in Table 4.

R_i	R_j	R_k
IsA	*IsA*	*IsA*
Implements	*IsA*	*Implements*
HasA:C_1	*HasA*:C_2	*HasA*:C_3[2]
IsA	*HasA*:C	*HasA*:C
HasA:C	*IsA*	Null[3]
Implements	*Implements*	*Implements*
Implements	*Uses*	Null
Implements	*HasA*:C	Null
ImplementedBy	*HasA*:C	Null
Generalises	*HasA*	Null
HasA	*Generalises*	Null[4]
HasA:C_1	*Part of*:C_2	Null
Uses	*HasA:C*	Null
Uses	*Uses*	*Uses*
Uses	*UsedBy*	Null
Produces	*InstanceOf*	Null
Updates	*InstanceOf*	Null
Removes	*InstanceOf*	Null
Uses	*InstanceOf*	Null

Table 4 – Entity Model Relationship Transformations

Implicit in Table 4 is the fact that only certain combinations of relationship pairs are valid. Removal of a software entity as a result of software evolution may produce ripple effects; these will be specific to the type of software entity concerned and are discussed further in chapters 6 and 7 on the effects of the removal of specific software entities.

2.3 The Removal of a Software Entity Instance

The removal of a software entity instance has implications for the consistency of the software instance model, quite apart from the ripple effects on any dependant software instances as discussed in the specific sections on removal of software instances in chapters 6 and 7. If I_y is the software instance to be removed then, for every pair $({}_xR^i_y, {}_yR^j_z)$ where ${}_xR^i_y \in$ parent relationships of I_y and ${}_xR^j_y \in$child relationships of I_y, map $({}_xR^i_y, {}_yR^j_z) \rightarrow {}_xR^k_z$ where (R^i, R^j) maps to R^k. The

[2] C_3 is a refinement of C_1 and C_2 (see Table 4).

[3] "Null" indicates that no mapping is required.

[4] This can, however, cause problems for data entity z which depends on data entity y, whose removal will have ripple effects on z. See chapter 6 section 5.1.1.

mapping $(R_i, R_j) \rightarrow R_k$ describes how pairs of dangling relationships are combined into single relationships, as shown in Table 5.

R^i	R^j	R^k
InstanceOf	$HasA^l$	Null
$HasA^l$	$HasA^l$	$HasA^l$

Table 5 - Instance Model Relationship Transformations

3 A Set of Software Entities and Software Instances

The term *software entity* is a generic term used to refer to any of the following concepts (or abstractions):

- A process (or thread of control - see section 3.2);
- A service (or functional capability - see section 3.1.2);
- A task (see section 3.1.1);
- A data mapping (see chapter 6);
- A message (see section 3.1.3);
- A DEM (see chapter 6);
- A DIM (see chapter 6);

amongst others.

A software entity is an abstraction, or modelling construct, useable by software developers when developing software systems. Furthermore, a software entity can be modelled in isolation from other software entities. This relies heavily on the abstraction of software entities and the assumptions that software entities make of other software entities on which they depend, since modelling requires that a software entity can be modelled in terms of abstractions provided by other software entities without having to know how those software entities are implemented. Every software entity is an abstraction which is parameterised in some way and, as such, is related to a number of software instances through an *InstanceOf* relationship.

Existing programming languages and models possess similar abstractions (such as functional and data abstractions) but implement them using different syntax and notation. This section describes a set of software entities which *build* on existing abstractions, whilst providing adaptability and an increased built-in tolerance to change.

Many software architectures restrict the set of software entities or embed them within other software entities, making individual software entities difficult to extract out of the software. Object-oriented models consist of three basic software entities:

- Classes;
- Methods;

- Control constructs [Booch91a].

There is no explicit modelling of software architecture knowledge and how the elements of software architecture relate to other software entities in the software. However, as shown in Figure 6, all software entities are software architecture components or connectors. Messages represent the connector aspects of a software system and allow a software engineer to express control-flow and data-flow in a software system (they are discussed in section 3.1.3). They are an important aspect of the architecture of a software system, in addition to component types such as those shown in Table 6. Notice that these component types are part of the architecture and not the domain. Hence, an operating system domain typically contains a scheduler component type, but this component type is a member of the domain and not the architecture which comprises the domain. Of course, individual architectures could be viewed as domains themselves, in which case the component types become members of the architecture domain.

Component Type	Architecture/Model
Scheduler	Blackboard
Event Controller	Event-Based Architecture
Shared Repository	Shared Repository Architecture
Class	Object-Oriented
Interpreter	Interpreter/Virtual Machine

Table 6 - Software Architecture and Component Types

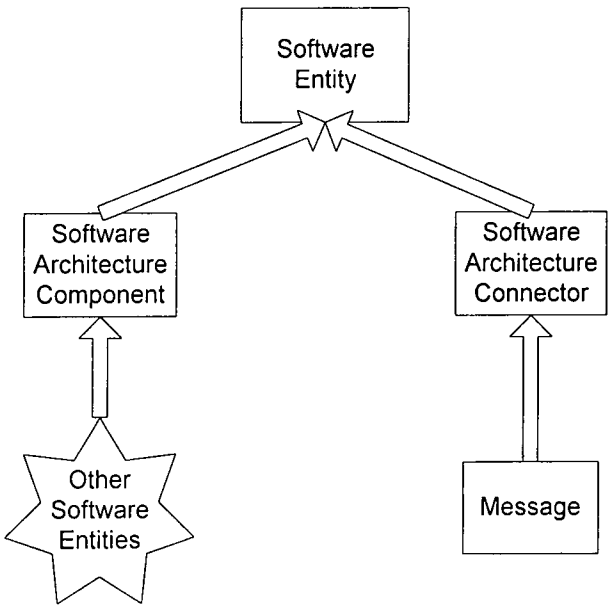


Figure 6 - Relationship Between Software Entities and Software Architecture

3.1 Functional Software Entities (FSEs)

Functional (or computational) software entities (FSEs) encapsulate the functional aspects of the software. They are of two types:

- Service software entities;

- Task software entities.

These two types of software entity are described in the next few sections, along with their inter-relationships.

3.1.1 Task Software Entities

A service software entity describes *how* to perform a particular task given a set of data parameters. A task describes *what* a set of services do, given a set of behavioural *and* data parameters. Hence, services provide a functional abstraction with respect to data, whereas tasks provide a functional abstraction with respect to data *and* behaviour. This distinction provides a separation of concerns between which task to perform and how the task is performed. A task encapsulates what to do. A clear separation is made between what is to be done (a task) and how it is done (a service) – see Table 7. In terms of the relationships in section 2.1, a service *Implements* a task.

Task	Service
Sort	Bubblesort, Quicksort, Heapsort
Determine reorder level	GetReorderLevel
Graph Layout	Spring Graph Layout

Table 7 - Example Tasks and Services

A task describes a set of services of a particular type or having a particular set of shared attributes, and how they relate to other sets of services when performing a particular task such as sorting. A service (such as a bubble sort, quick-sort, graph layout algorithm) is an implementation of a particular task and is assumed to only evolve in well-defined ways. For example, if service A calls service B, and B uses data structure C as a parameter, and C changes, then A's C parameter to B must also change. This is a change as a result of a ripple effect. In other words, services are used as-is and take the form of a well-defined algorithm that can be re-configured only by changing its set of parameters. This modularisation is evident in LISP, where generic functions are used to represent a generic function such as sort, and methods are used to represent a specific implementation of a generic function that has the same name, with an appropriate specialisation of the generic function's parameters [Steele90a]. This type of modularisation is useful when polymorphism needs to occur on more than one of the generic function's parameters; in traditional object-oriented languages, polymorphism can only occur on the object to which the message is sent i.e. only to the first parameter of the method where the first parameter is assumed to be the "this" or "self" parameter. Hence, using this terminology, tasks are equivalent to generic functions and services are equivalent to methods.

The task/service separation of concerns is similar to the generic function and method separation of concerns in LISP. Tasks, like generic functions, provide an interface that consists of the most general set of parameters required for all the services that implement the task. Each relevant combination of parameters is then given an implementation in terms of a particular service.

The task-service separation of concerns provides for extensibility by allowing the software engineer to express dynamic binding of functionality through the provision of a larger context for service calls, much like Seiter does for object-oriented software [Seiter98a]. In Seiter's approach, method dispatch is based on both the class and the identity of the

target object, as opposed to standard object-oriented models in which method dispatch is based on just the class of the target object. This allows behaviour to be customised on a per-object basis. In other words, the mapping from task to implementation of the task is more parameterised. The task-service separation of concerns permits limited extensibility of behaviour, which must be within the interface of the task.

The separation of task and service (or implementation) knowledge allows one to change either without affecting the other. For example, assume that service S requires a particular task "Sort" to be performed. The particular service to which "Sort" is related through an *ImplementedBy* relationship is "BubbleSort". The appropriate service can also be dynamically bound to the task at run-time through the use of a broker component. This is essentially the idea behind facilitators [Genesereth94a] and the work of Skarmeas et al [Skarmeas97a], both of which emphasise the centralised (or brokered) routing of messages. Skarmeas et al use a "message board", a centralised component that maps messages from clients to appropriate servers based on the content of the message. The two approaches to interpreting the content of messages are:

- Templates;
- Active Patterns, or functions that accept the message as a parameter and provide a semantic test based on the content of the message [Skarmeas97a] (see also chapter 7 section 3.2.3.1).

The main characteristic of these approaches is the centralised nature of the mappings between the task knowledge and implementation (service) knowledge.

The reason for the separation of task knowledge from service knowledge is localisation of evolution. If the relationship between which task to perform and how it is performed changes, then a simple change to the broker's knowledge store will allow all clients of the particular task to be updated with this knowledge, improving the adaptability of the software. Compare this with traditional software in which the task and service knowledge is lumped together. A change in this relationship means changing every single occurrence, which may be a costly task. The problem lies in determining the appropriate (task, service) tuples when the software is designed and not falling into the trap of making these mappings implicit.

Assuming that the change is such that the mapping between task and service changes (rather than the task or service content changing), which indicates that a different implementation of the task is required because the environment has changed or a better implementation exists, then the mapping can be changed without affecting the client (the service that invokes the task). This means that such a change has a limited effect on the client, and evolution is therefore more localised.

3.1.2 Service Software Entities

Services encapsulate functional abstractions with respect to data, and are similar to functions and methods in traditional programming languages. Figure 7 shows the relationship between services, messages and data. Figure 8 shows some specific relationships that exist between services and tasks. A service such as BubbleSort *Implements* the task Sort. A

task such as Sort *Implements* task Filter. A service consists of a sequence of messages, which encapsulate calls to services. This comes from the observation that all services consist of a sequence of service calls, even programming language constructs such as conditionals and loops, which can be represented using prefix function notation (as used in Lisp, as can be seen in Figure 9).

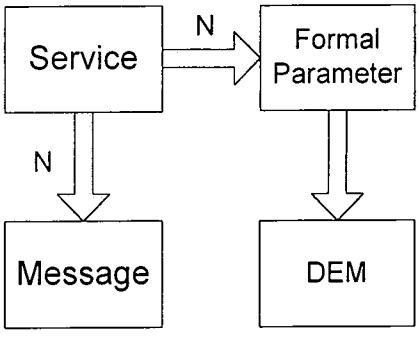
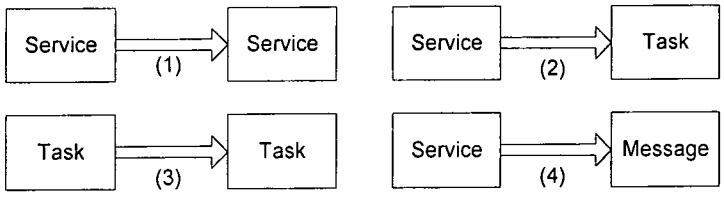


Figure 7 - Service Entity Model



KEY

(1) No relationships.

(2) *Implements*

(3) *IsA*

(4) *Calls*

Figure 8 - Relationships Between Services, Tasks and Messages

As an example, consider the BubbleSort service shown in chapter 6 figure 3. This can be represented as a sequence of tuples[5]:

BubbleSort (DEM Sort) = [(Service = "For", Start = "Size (Sort.Record) – 1", Stop = "2", Step = "-1", Body = "BubbleSort'")]

BubbleSort' (DEM Sort) = [(Service = "For", Start = "0", Stop = "LoopVar", Step = "1", Body = "BubbleSort''")]

BubbleSort'' (DEM Sort) = [(Service = "If", Cond = "Sort.Rec[LoopVar] > Sort.Rec[LoopVar+1]", Action = "Swap (Sort.Rec[LoopVar], Sort.Rec[LoopVar+1])")]

[5] Square brackets "[…]" represent a sequence. Parentheses "()" represent a tuple. The representation is a sequence of service calls (represented as n-tuples).

which, for clarity, is a simplified (and self-explanatory) version of the software entity model representation in which messages have been taken out, and replaced by a simple language which can be converted to the appropriate software entity model. Nevertheless, it shows how a service can be represented by a sequence of service calls, and how loops and if statements can be represented as services.

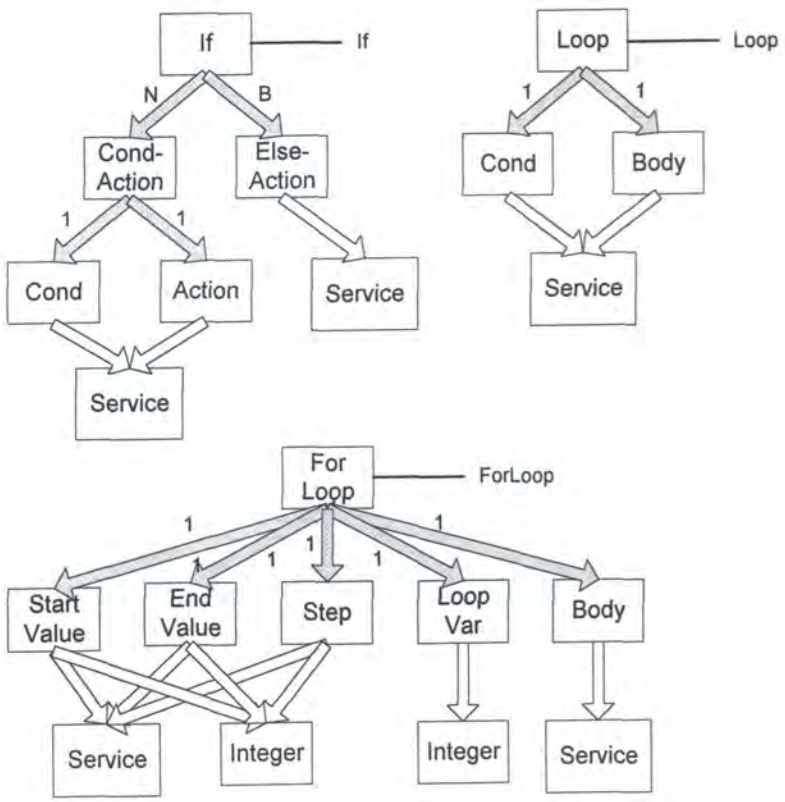


Figure 9 – "If", "Loop" and "For Loop" Software Entity Models

The modelling of services in this way provides in-built flexibility to potential changes in the calling architecture of an application; changes from, for example, a local procedure call to a more complex remote procedure call can be performed through a well-defined interface to the messages that make up the service body. This is possible because the assumptions about calling have been lessened to that of deciding between a set of calling architectures. Of course, any software can be made more flexible in this way providing that all assumptions can be extracted out as a parameter, with built-in mechanisms that allow the parameterised software elements to be changed. This also assumes that all potential adaptations have been identified and an appropriate mechanism for implementing that adaptation provided. But, as Len Bass et al point out, it is impossible and impractical to be able to determine all assumptions being made [Bass98a], even though particular assumptions can be identified and given this treatment.

These assumptions come in many forms:

- Assumptions about parameters – their number, structure and types;
- Assumptions about which services can be called;
- Assumptions about where these services reside;
- Assumptions about architectural characteristics, which is part of the design of the system i.e. what calls what etc.

Evolution of services in traditional software typically occurs as a result of a change in assumptions on which the service depends. For example, a software system uses a graph layout algorithm which assumes that the graph is a 2D graph. If this graph changes to a 3D graph, then the algorithm will be invalid with respect to the new graph and the software engineer either has to adapt it or create a new algorithm.

Any non-trivial software system will consist of both primitive and non-primitive services. Primitive services are characterised by the following characteristics:

- They encapsulate a well-defined, unchanging algorithm. For example, a graph layout algorithm, or bubblesort. They don't evolve because they are well-defined, static services;
- They arbitrate access to an entity in the software system's environment, which doesn't change. For example, hardware or processes.

The mapping between tasks and services is:

Task T (x_1 x_2 x_3 ...x_n)
Service S_i (x_{i1} x_{i2} x_{i3} ...x_{in})
where Name (S_i) = Name (T) and x_{ij} *IsA* x_j

Figure 10 - Mappings Between Tasks and Services

Hence, the mapping between a task and its service implementations is determined by both the data and behaviour parameters. Behaviour parameters are similar in some respects to Quality of Service (QOS) parameters in networking, where the user of the software or the software itself is able to choose the behaviour of the software based on these parameters [Tanenbaum96a]

Services are constrained to use data that's only accessible as a formal parameter to the service; there is no concept of global variables. Object-oriented methods can be modelled as services by considering an object of a class as the first parameter to the method, an approach that is often used to model classes in a non-object-oriented language, and often termed the "this" parameter. The integration of a service with another service in the context provided by a process is performed by inserting a service call in the form of a message somewhere in the service. At this point, it must be decided which data in the state space[6] is used as actual parameters to the service. This means that services are developed in terms of formal parameters, which encapsulate their data requirements. Their integration into the system requires them to be linked to the actual data structures of the system.

Batory suggests extracting complexity from components by taking a minimalist approach with simple components, and using generators which encapsulate knowledge of how to combine components [Batory93a]. He is interested in making software libraries less complex by lessening code repetition in them. Code repetition arises from the inclusion of such features as:

[6] The seeming contradiction between the state space concept and the stated lack of global variables can be explained by the following: services are expressed in terms of their formal parameters which are ultimately derived from the process state space at run time.

- Concurrency;
- Iterators.

in software libraries, features which can be extracted out into generators. This is also the aim of work on co-ordination languages [Gelernter92a], separation of concerns, and of the modularisation into software entities in this work.

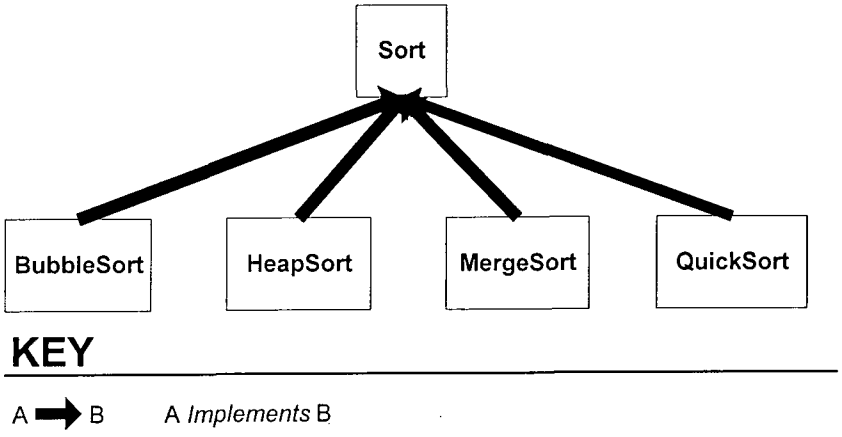


Figure 11 – Sort Task and Service Model

The functional aspects of all services are expressed in terms of formal parameters. There is no concept of global variables. Formal parameters are linked to actual parameters in the state space of a process at service integration time. Access to state space variables is through an interface consisting of four services:

- StateSpace_Produce (DIMName, NewValue) – allows the software to signify production of data; creates a *Produces* relationship;
- StateSpace_Get (DIMName) – allows the software to access *existing* data; creates a *Uses* relationship;
- StateSpace_Remove (DIMName) – allows the software to remove data from the software system; creates a *Removes* relationship;
- StateSpace_Update (DIMName, NewValue) – allows the software to update the value of data in a software system; creates an *Updates* relationship.

This provides a well-defined interface for variable use, a mechanism that is usually hard-coded into traditional high-level languages. Using this approach allows the software to reflect upon *data use*, and utilise this information to determine ripple effects and improve particular aspects of the adaptability of services, as discussed in chapter 7.

3.1.2.1 Service Interface Attributes

There are a number of important aspects to a service's interface in addition to formal data parameters, on which clients of the service depend. These relate to the behaviour of the service and the relationship between the service and data.

First of all, consider the behaviour of a service. Semantically, this is a difficult aspect to model. However, there are characteristics of behaviour:

- Which can be extracted out from a service and represented using a reflective mechanism, and;
- On which clients of the service depend.

These characteristics of behaviour (as shown in Table 8) can then be represented in the reflective software entity model as "interface attributes". By linking changes in the service to changes in these "behaviour characteristics", the effects on the service's dependants can be determined more easily and with more clarity.

Behaviour Characteristic	Description
Duration	The time it takes the service to execute.
Space Requirements	The amount of memory the service uses.
Requirements	Represents the behaviour as a whole and relates it to the requirements which it implements. As will be seen in chapter 7, this allows changes in the behaviour to be related to their effects on how well the behaviour satisfies what is required of it.

Table 8 - Service Entity Behaviour Characteristics

With respect to the relationship between service and data, there exist three important types of relationship:

- Data is P*roducedBy* services, Services *Produce* Data;
- Data is R*emoved* from scope, Services *Remove* Data;
- Data is U*sed* by services, Services *Use* Data;
- Data is *UpdatedBy* services, Services *Update* Data.

These relationships are used in conjunction with the call-graph and the start state space of the process to determine actual parameters for a newly-integrated service. The *Update* relationship means that the service updates the values of the data instances in the DIM; changes don't include evolution of the DEM.

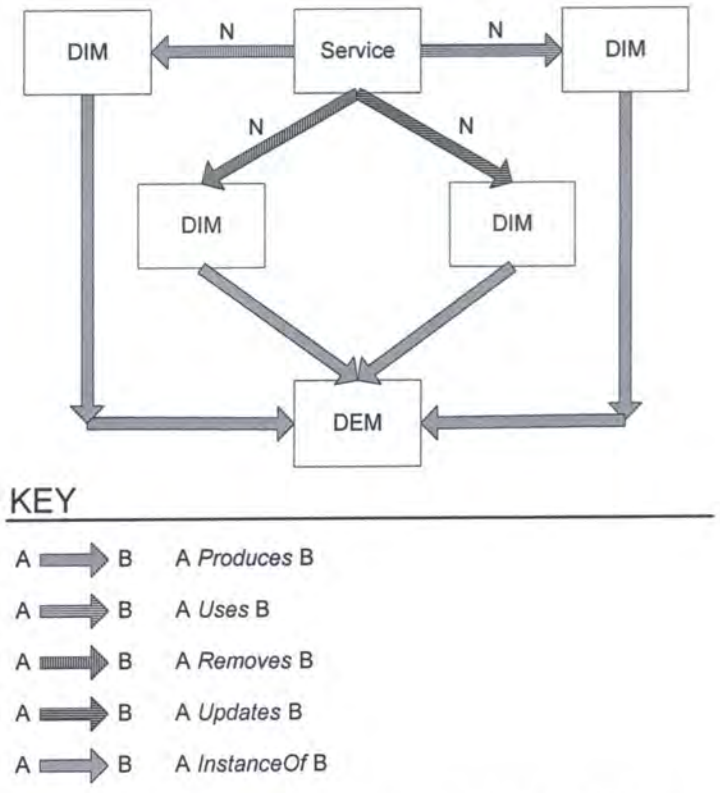


Figure 12 - *Produces*, *Uses*, *Removes* and *Updates* Relationships

A service can be analysed to determine what these relationships are for the service, and how changes to a service can affect these relationships. For example, before a change, a particular service may produce data A, whilst after the change it no longer produces A. This has an effect on the processes' state, which in turn has an effect on any services in the execution tree that are executed after the particular service. The reflective model can help determine this, without having to resort to a tedious repetitive re-compilation process to determine problems with missing parameters, as is currently the case.

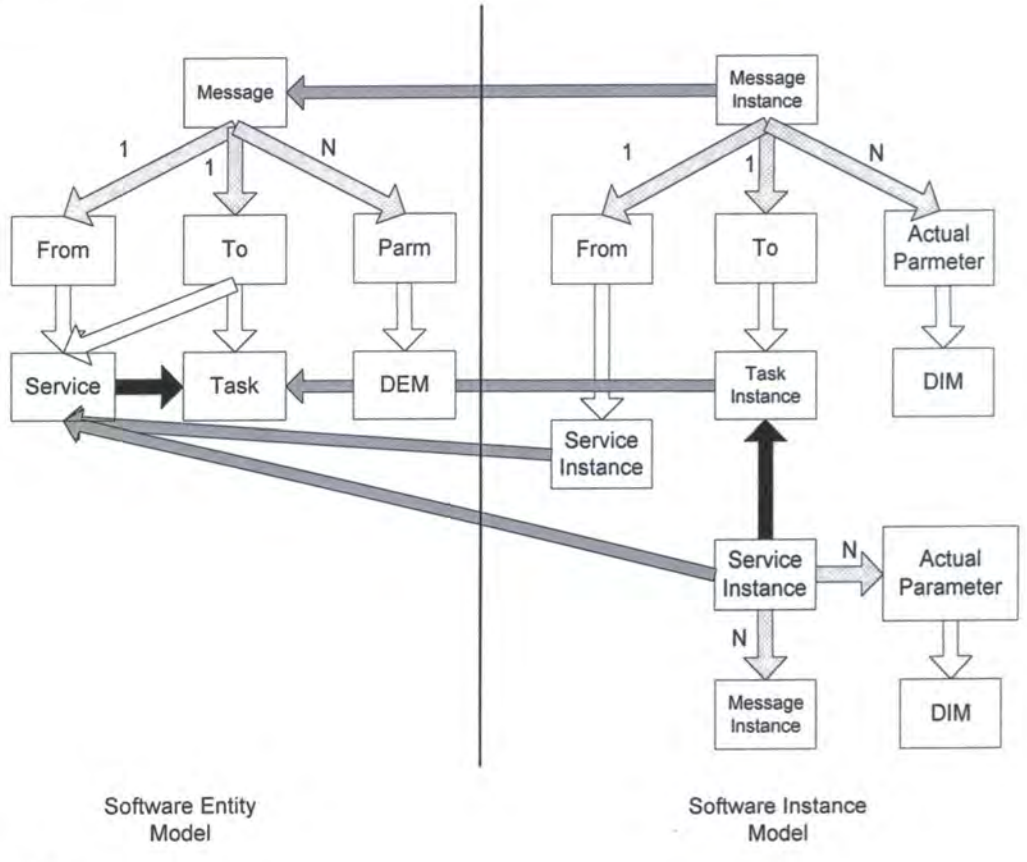
Whereas services represent behaviour in a fairly context independent manner, service instances represent behaviour in a particular context i.e. that of process and data instances. A service instance is related to a service through an *InstanceOf* relationship. Whereas a service is defined only in terms of its formal parameters, a service instance is defined in terms of actual parameters and is linked into the call graph of the particular process (see section 3.2) in which its behaviour executes.

3.1.3 Message Software Entities

Message (or service request) software entities encapsulate information about which task to call and the formal parameters (both data and behaviour) to pass to the service. They are similar to Smith's external object descriptions [Smith93a], which instantiate a function or choose a subset of the capabilities of a function through its interface. Messages can also be termed Service Activation Records (SARs), a term that is similar to Knowledge Source Activation Records (KSARs) in the blackboard literature [Brownston95a]. In effect, this distinction recognises that service execution is different than the service itself and so service execution (or service activation) should be reified in its own right. This is the objective of the message software entity.

In traditional software, messages aren't reified and therefore not manipulate-able by the software. By reifying them, the software or software engineer can manipulate them directly in order to perform evolution. In addition, by modelling their relationships with other software entities and, by appropriate modelling of these dependent software entities, the effects of this evolution can be determined.

Figure 13 shows the structure of a message and how it relates to FSEs. In essence, the message provides information on which task or service the "From" service is requesting to be performed. In particular, notice that "To" can be either a task or a service. Normally, "To" should be a task so that the advantages of the task-service distinction can be used. However, if it is known that the "To" service will always provide the required capability and so the dependence never breaks, there is no need to go through the extra level of indirection provided by the task abstraction. This is primarily an efficiency-improvement measure. A message also encapsulates actual parameters from the process state space to be passed to the target service.



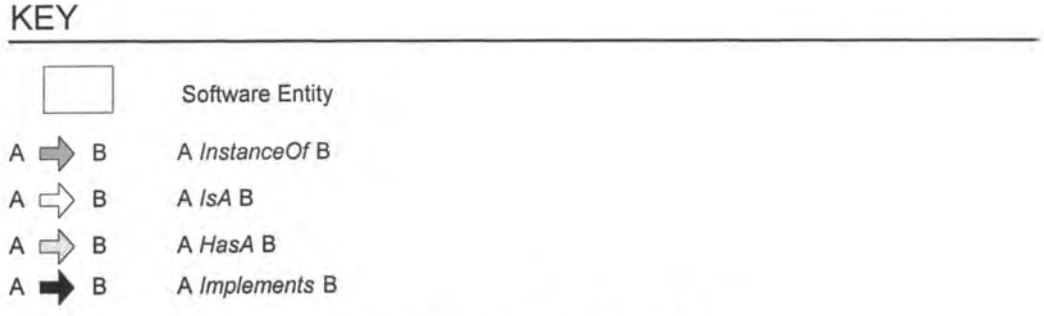


Figure 13 – Message Entity and Instance Models

The procedure to follow when a service requires new data and any service instances which are an *InstanceOf* the changing service therefore need to be updated, is:

- Create a new DIM which satisfies the data requirements of the service;
- Create a *Uses* relationship between any service instances which of this service, and the DIM;
- Trace back through the call graph to find a service instance that can produce the DIM;
- Create a *Produces* relationship between this service instance and the DIM.

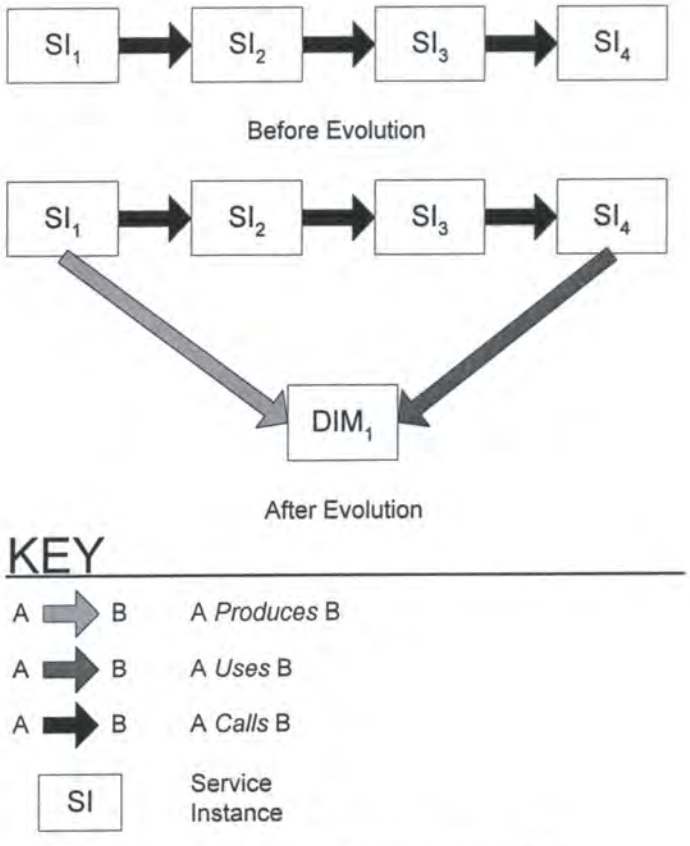


Figure 14 - Modelling Actual Parameters in SEvEn

Meek proposes a procedure call abstraction in [Meek95a] in order to afford procedure call compatibility between different programming languages. His abstraction is based on picking out the important abstractions in any procedure call mechanism:

- The caller;
- The callee;
- A request of the callee. The form of this aspect of a procedure call is dependent on the form of the callee: if the callee provides a wide range of functional capabilities, then the request must encapsulate the particular functional capability that is required; if the callee provides only one functional capability, then the request is implicit in the fact that the callee is being called.

In addition, the following characteristics determine the form of the procedure call:

- Is a result returned by the procedure call and, if so, when is the result returned? What is the form of the result; is it data or is it a control-based result such as an acknowledgement of some event? In addition, is the call synchronous or asynchronous?
- Does the procedure call have any actual parameters?
- What is the semantics of the parameter-passing: call-by-reference, call-by-value, local procedure call, remote procedure call etc.?

Different programming languages have different support for these mechanisms, implement them in different ways, provide a different syntax for their representation, and provide a different syntax for different forms of procedure call. For example, asynchronous procedure calls are not *directly* supported in most programming languages, but must be implemented in terms of the basic building blocks provided by the language. There is no underlying, common mechanism for the representation of all forms of procedure call semantics.

There are a number of important aspects to messages, in addition to their use as a service execution reification abstraction. These include:

- Visibility;
- Message types.

Visibility deals with the endpoint of a message i.e. which tasks the message can use, and ultimately which services a particular service can call. Many software architectures attempt to limit or constrain "functional visibility" in order to provide some semantics to the functional aspects of software by saying which combinations of functions are valid and which aren't. For example, an object-oriented approach can be used to model a filter architecture consisting of an input module, a filter module and an output module by using classes to model each module and constraining the control flow to utilise an object of each type in turn. Hence, at each stage of the control flow, the functional visibility is constrained to use methods belonging to a particular class. This is a useful semantic mechanism for ensuring valid designs with respect to a given architecture, but can cause problems when new requirements conflict with the assumptions built in to the model.

The message type provides a way to characterise a message based on a number of attributes or characteristics shown in the rows and columns of Table 9, along with a number of examples.

	Local	Remote
Synchronous (blocking)	C, Pascal, RPC	RPC
Asynchronous (non-blocking)	Semaphores	CORBA Message

Table 9 - Message Characteristics and Types

Message characteristics form an evolution space for messages. A particular type of message implementation can be chosen by passing a parameter to the message when it is reified. This provides an advantage over traditional software

languages which are typically tied to one particular message-passing style, which is difficult to change. For example, the following code assumes synchronous, local call semantics:

```
void X (...) {
        ...
        int i = sort (Data);
        ....
}
```

This can be expressed as the behaviourally-equivalent code:

```
Message M = new Message (X, sort, Local, Synchronous);
M.send ();
...
int i = M.getReturnValue ();
```

without invalidating the requirements, and it provides the ability to change the type of message required because it doesn't make any assumptions about the message characteristics identified in Table 9. The assumptions are essentially moved into the interface of the message software entity, an approach that has its origins in the open implementation approach [Kiczales97b], and the interface made as generic as possible by choosing the interface that makes the least number of assumptions.

3.1.3.1 Message Conflicts

Message conflicts within a service occur when the following are true:

- Two or more messages within the service both update or remove the same resource or DIM;
- One of the messages is part of the existing software system, the other message has been added as a result of evolution of the service.

For example, a sort service consists of a set of messages M_1, M_2, ..., M_i which use resources R_1, R_2, ..., R_j. Evolution results in the addition of a new message, M_k, which updates resource R_2. This means that M_k interferes with a resource on which existing messages depend, hence causing a potential conflict in the service as a whole (see section chapter 7 section 3.3.2 p249 for how SEvEn copes with this).

3.2 Process Software Entities

Peterson et al state that:

> "A key attribute of a good architecture is the separation of the coordination strategy or model (the flow of control through the software, or mission) from the providers of operations or services (the building blocks or components). This separation:
>
> - Allows a change in operation, or service provided (potentially due to a new piece of equipment or information), without requiring a change in the existing mission software, and;
> - Allows a change in the mission without necessarily requiring a change in the service providers carrying out that mission." [Peterson94a p13]

However, control and co-ordination information is typically difficult to extract from software into separate control/co-ordination constructs because of the high level of coupling between control and capability, a by-product of functional abstraction. Hence, services have control aspects and control constructs call services. This high level of inter-dependence leads to difficulties in extracting out the control. Lieberherr and his research group at Northeastern University attempt to overcome this limitation by trying to extract out the control aspects of object-oriented software using an abstraction called propagation patterns, which are constructs that specify how to "walk" an object graph [Lopes94a, Lieberherr96a]. These propagation patterns call "visitor methods" (methods attached to classes) along the way. A disadvantage of this approach is the limited modelling power of the propagation pattern abstraction and, in particular, the inability to represent conditionals in the propagation pattern abstraction. Others have also argued for a separation between computation and co-ordination, for example Gelernter and Carriero [Gelernter92a], even though it may not always be possible to separate out co-ordination and computation in all domains.

Functional software entities provide functional abstractions i.e. they parameterise the "behaviour space" of the functional abstraction (the set of functional capabilities that the functional abstraction is able to produce) using data and behaviour parameters. The process software entity, however, provides a process abstraction which allows the software engineer to represent information regarding co-ordination between functional abstractions (FSEs) i.e. which FSEs are called, when, and which data is used as their formal and actual parameters.

The process software entity provides a context for a thread of control which comprises a call graph consisting of services and messages. Services encapsulate context dependent behaviour, whereas messages represent the flow of control between services and are akin to messages in object-oriented systems and procedure and function calls in other languages, such as C and Pascal.

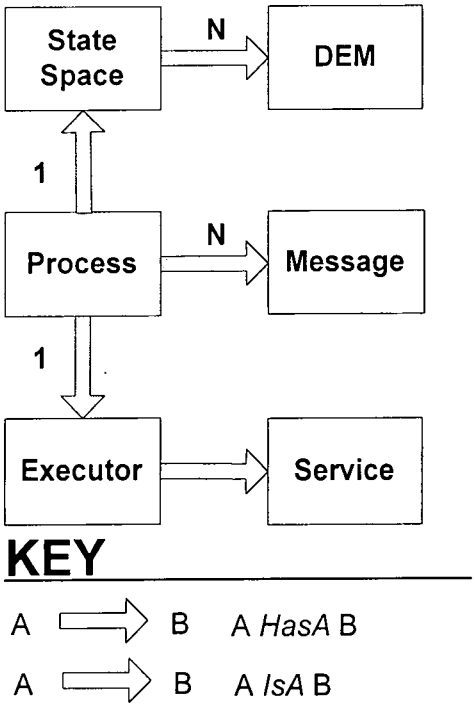


Figure 15 – The Process Software Entity Model

As described in section 3.1, services and tasks encapsulate the functional capabilities of the software. The processes in a software system provide a modelling construct that allows co-ordination of services to be specified and formal parameters to be linked to services using direct relationships. The format of a process, shown graphically in Figure 15, is:

Process ::= (Call Graph, StateSpace, Executor/Interpreter)

The Call Graph represents the flow of control through the FSEs in a software system (for example, see Figure 16), and is dynamic in nature due to the loose coupling between services which is determined at run-time based on behaviour parameters. The StateSpace consists of a set of DIMs which provide actual parameters to the service instances in the Call Graph. In addition, the StateSpace has a start state, the set of DIMs that form the StateSpace before the process is invoked. In short, processes provide a context, consisting of data in the form of a state space of DIMs, for the execution of service instances.

Any software system must contain at least one process, into which services, messages, tasks and DEMs are integrated. Process software entities impose rules on the integration procedure to ensure service expressivity and parameter adaptability.

The granularity of a process is determined by the level of abstraction of the sensors and effectors (services, see section 3.1.2) it uses. The level of abstraction of services is determined in part by the level of abstraction of services that they use. This recursive definition means that a primitive service is less abstract than a non-primitive service, and a non-primitive service expressed in terms of other non-primitive services is more abstract than these other services. For example, a process that uses sensor services which are expressed in terms of the software system's environment is of higher granularity than a process that uses sensor services expressed in terms of an internal software system environment

such as a data structure. In the first type of process, control is system-wide and changes to control occur at this level of granularity whereas in the second type of process, control is more distributed and changes to control are therefore distributed.

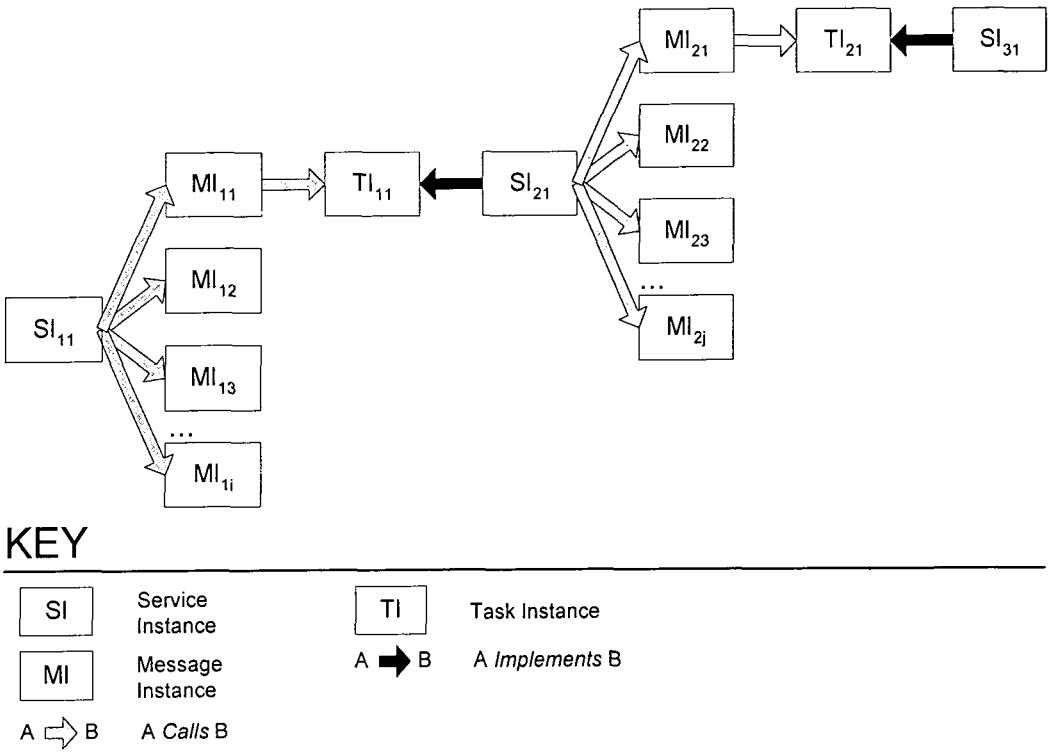


Figure 16 – Process Call Graph Structure

There is a question of whether the behaviour of the software should be driven by the system environment or by each individual software entity's environment. This has consequences for the level of granularity at which processes should be used. If behaviour is driven by the system's environment then processes are limited to those software entities whose environment consists solely of non-software entities i.e. their environment is purely external to the system. No other software entities are allowed to possess a thread of control and autonomy is very limited. If, however, behaviour is driven by each software entity's environment, then each software entity is potentially allowed to possess a thread of control and therefore execute a process. Hence, there are more processes and more autonomy. The disadvantage is more complex software that is probably harder to design. In addition, such an approach (of making functional components as autonomous as possible) is not always the most natural way to model a problem. As an example, consider a sort program consisting of input, sort and output services. Each is a potential target for autonomy i.e. each of these three components could be given a control thread in the form of a process. This, however, isn't a natural model for a sort program, which is better modelled using a single process which calls the three components in turn at appropriate points.

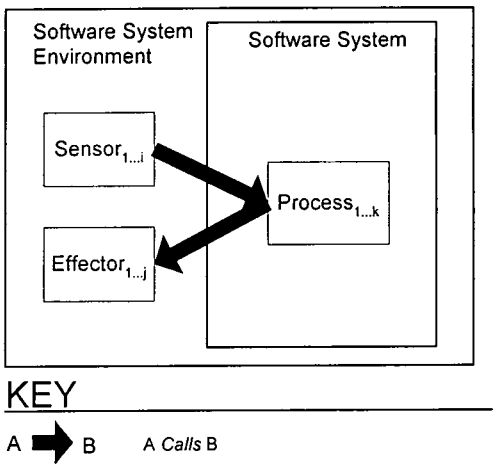


Figure 17 - Processes and Their Environment

The distribution of control knowledge is an important characteristic of software systems. Some software systems are inherently constrained to a particular pattern of control. For example, a sort program generally has a centralised thread of control which ensures that the input, sort and output services are called in order. Control could be shifted to each service, creating three autonomous components, without affecting the functional requirements. This, however, is an unnatural way of modelling the problem. Another example is a spring graph layout problem, which can be modelled using either a centralised or distributed control approach. A centralised approach utilises a single thread of control, or process, which encapsulates the centralised control knowledge (Figure 18 (b)). A distributed approach gives each graph node a thread of control (Figure 18 (c), represented by "NodeLayout"), which ensures that the node is laid out with respect to the other nodes in the graph. The data entities (Graph, Edge and Node) are the same, but the structure (the relationships) is slightly different although both represent the same information. The algorithms are different also, even though the resultant behaviour is the same.

Passive FSEs represented by a service or sequence of services can be converted to autonomous, active components by converting the group of services into a process. This conversion requires encapsulating the services within the process and adding a state space to the new process software entity. It also requires the introduction of more services, depending on the type of conversion which, in turn, determines the new processes' interaction with its environment. The conversion type is determined by the type of sensor service (see Table 10) attached to the new process.

Sensor Type	Environment	Comments
Timer	Time, timer	The timer sensor service is interrupt driven, and calls the converted services at periodic intervals driven by the environment. It could be argued that the process is not active because it is driven by another active component, the timer, but the timer and new process are different processes, and the new process executes independently.
Data Change	A set of data structures	The sensor detects a particular change in a particular set of data structures and invokes the process when such a change occurs. This pattern is similar to blackboard knowledge sources

		[Corkhill91a].
Temperature Sensor	Temperature	The sensor is a hardware sensor, and the process is invoked when a particular condition is met in the temperature sub-environment. For example, when the temperature drops below a particular level, the process is invoked.
Continuous	Any	This type of sensor is essentially null, because the process executes continuously and is not driven by its environment. It is still active, but only affects its environment (it isn't affected by its environment).

Table 10 - Sensor Types

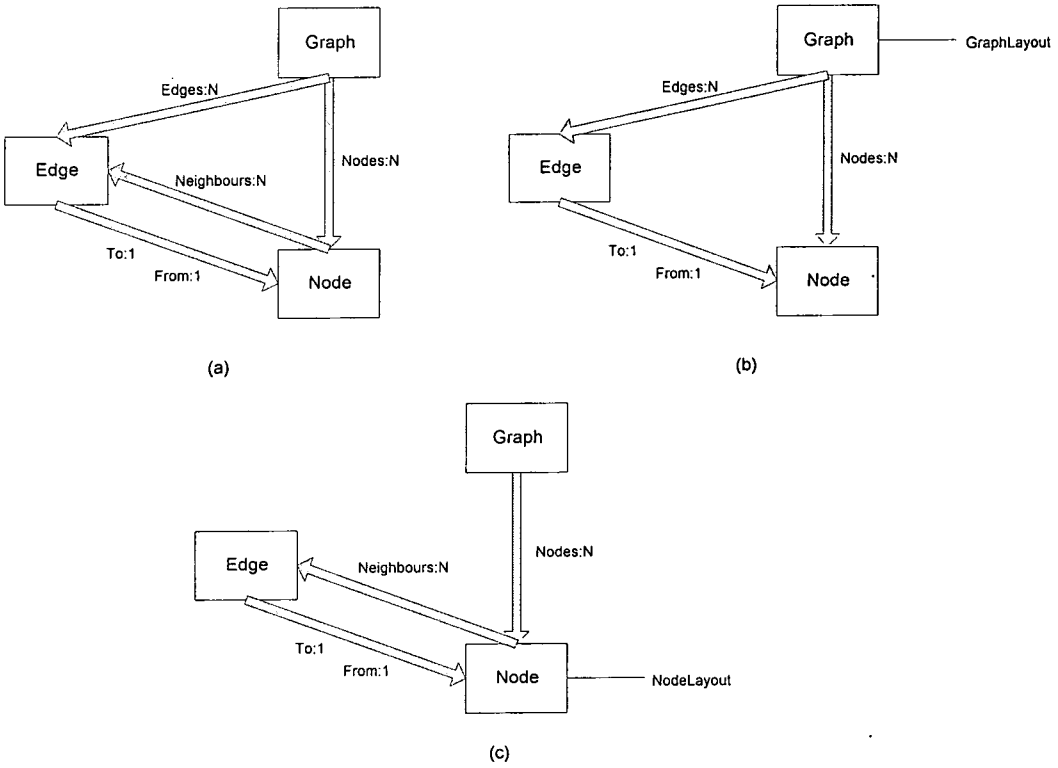


Figure 18 - Evolving Graph DEM

3.2.1.1 Call Path Software Entities

A call path is a unique path through a call graph, for which the call path software entity provides an abstraction.

3.3 Data Software Entities

Data is represented within SEvEn's conceptual framework using a form of entity-relationship model, the DEM, which is described fully in chapter 6. Data conversions, an abstraction which encapsulates data conversion information, are also described in chapter 6, along with data instance models (DIMs).

3.4 Software Entities and Software Entity Instances

Every SEvEn software system consists of a set of software entities and software instances. Each software instance is related to exactly one software entity through an *InstanceOf* relationship, and to other software instances through *HasA* relationships.

The existence of a distinction between software entities and software entity instances is an inherent part of any software entity because a software entity is an abstraction with respect to some characteristic, and the software entity instance provides a way to instantiate the characteristic, as shown in Table 11. This distinction is typically not present in traditional software languages and models, even though it is a useful technique for increasing the modularity of the software model, and hence increasing the "connectedness" of the model so that a better analysis of adaptability and ripple effects can be performed.

Software Entity	Software Entity Instance
DEM	DIM
Service	Service Instance
Message	Message Instance
Task	Task Instance
Process	Process Instance
DEM Path	DEM Path Instance
DIM Path	DIM Path Instance
DEM Mapping	DEM Mapping Instance

Table 11 - Software Entities and Software Entity Instances

Figure 19 shows how software entities and instances are related to each other. The difference between the instances above and the software entities of which they are an instance is that, whereas services, messages, tasks and processes deal with DEMs, service instances, message instances, task instances and process instances deal with DIMs.

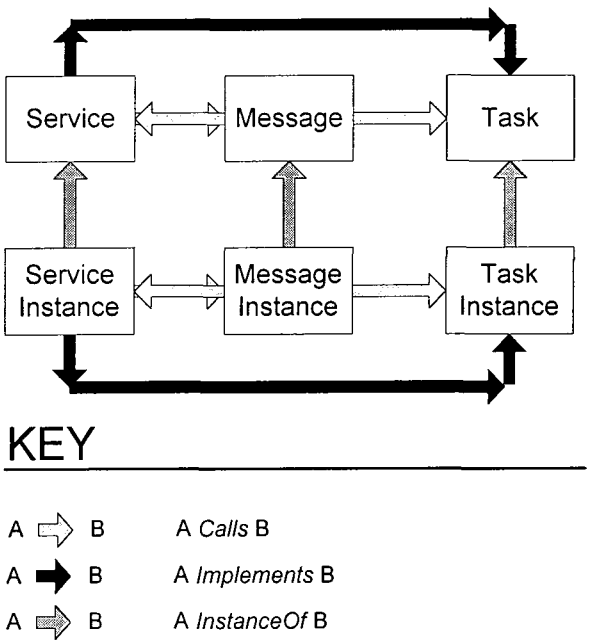


Figure 19 - Relationships Between Software Entities and Instances

3.5 Instance Evolveability

The coupling between instances and software entities is very strong, making it difficult to improve the adaptability of instances with respect to software entities. Hence, changes in software entities will typically produce ripple effects in any instances. However, these ripple effects are typically known because of the special relationship between instances and software entities provided by the *InstanceOf* relationship. This relationship means that changes in software entities are typically mirrored in their instances, which means that, if the change in the software entity is known, the required change in the instance is also known. For example, a change in a DEM can be directly mapped to a change in any instances of the DEM. This is helped by the identification of change types for software entities, so that changes in software entities can be expressed in terms of a sequence of change type applications, which can be mapped to change types in the instances of the evolving software entity. This may result in loss of information, but this may be a desired side effect of the change anyway.

3.6 Software Entity and Software Instance Paths

A software entity path is of the form:

$$\text{Software Entity Path} ::= \langle(\text{SoftwareEntity}_i,\ \text{parent}|\text{child}\ \text{relationship-type}\rangle$$

where <...> indicates a sequence, SoftwareEntity_i refers to an entity in the software entity model and the "|" character designates "or".

A software instance path is of the form:

Software Instance Path ::= <(SoftwareInstance$_i$, parent|child relationship-type>

where <...> indicates a sequence, SoftwareInstance$_i$ refers to an entity instance in the software instance model and the "|" character designates "or".

3.7 Representation of Software Architecture

Software architecture is an holistic characteristic of software systems, a characteristic that can't be represented as a software entity, and so software architecture is rather like a "distributed component". There are two orthogonal aspects to software systems with respect to software architecture:

- Levels of abstraction, or vertical structuring;
- Modularisation within levels of abstraction, or horizontal structuring.

Both are modelled using the relationships described in section 2.1. The difference lies in the parent and child of the relationship. So we have:

- *HasA* – data abstraction;
- *IsA* – data abstraction and task abstraction;
- *Calls* – functional abstraction with respect to behaviour and data (because a service calls a task, which is an abstraction with respect to behaviour and data);
- *InstanceOf* – abstraction;
- *Implements* – functional abstraction with respect to behaviour.

Horizontal structuring is equivalent to software architecture, and is concerned primarily with FSEs and the *Uses* relationships that exist between FSEs which call each other. Software architecture is a property of a software entity model (representing anything from a single object to a whole software system) as a whole and is thus implicit in the model. This subject will be returned to in chapter 7, when FSE adaptability with respect to software architecture is considered.

Even though software architecture can not be represented as a software entity, the mapping between software architecture and the software entities of a software system is known. Garlan and Shaws' work [Garlan93a] allows a software system to be analysed and its software architecture to be determined by representing the software as a set of inter-related components and connectors. In SEvEn, each software entity is either a component or a connector, as Figure 6 shows. The connectors are an important aspect of software architecture, because it is the patterns of communication in a software system represented by messages that determines the software architecture along with the types of components. For example, a blackboard architecture constrains its components to be either:

- Domain knowledge sources, consisting of a trigger condition, a pre-condition and an action;

- Control knowledge sources, which also consist of trigger- and pre-conditions and actions, but which represent the control knowledge of the application.

The connectors are also constrained: they consist of reads from and writes to the blackboard data structure which contains the application data. Hence, as section 3.1.3 describes, message characteristics such as the from, to, and content attributes are important in determining the patterns of communication, which are in turn important for determining the software architecture of a software system.

4 Summary, Discussion and Conclusion

The completeness of the software entities introduced and described in this chapter can be questioned due to the fact that no major analysis has been carried out to try and exhaustively identify all the mutually exclusive software entities of any software system that can be modularised or extracted out. The argument is that the major elements have indeed been identified that will form the bulk of the evolution in a software system. Other software entities may exist, indeed an entirely orthogonal taxonomy to the one described in this chapter may exist, possibly at a different level of abstraction. For example, one could abstract out an event processing subsystem in a software application that has a user interface and apply the principles described in this chapter. However, that is a particular example of the theory described here, which can be applied to any such set of software entities and identified relationships.

The identification of higher level software entities has ultimately been a failure because there is no well-defined way of adapting them i.e. the adaptation interface is different for each of these software entities, so there is no way to capture knowledge about how changing a software entity affects other software entities that depend on this changing software entity. The whole idea of developing a set of software entities is to determine how these can change, and how these changes affect other software entities that depend on them. Higher level software entities have to be treated as special cases, with a set of adaptations developed for each one. However, this contravenes the assumption that one can't predict how a software entity will change. The choice of lower level software entities described in this software is a consequence of this. Hence, these software entities in particular were chosen because of their stability and reusability i.e. the fact that they are domain independent.

Some software entities provide a more abstract modelling construct than others. Even different types of a software entity may be at different levels of abstraction, because the components and connectors are abstract and don't constrain the types of component and connector allowed. For example, a blackboard architectural style says very little about the characteristics of the components, apart from that they should be condition-action pairs. A sort architecture, on the other hand, is a type of filter architectural style with the constraint that there must be three components; input, sort and output, which must have particular well-defined characteristics.

The set of software entities described in this chapter include the reification of messages. This begs the question of how services communicate with messages in order to request another service to perform a task. This communication can take the form of messages with local, synchronous semantics, much like procedure calls in programming languages. These

messages are not reified, which means that they can't be manipulated or used by the base-level software entities, unlike the message software entities described in section 3.1.3.

The abstractions utilised in modern programming languages are there because they have "evolved" over the years, created by software engineers who have a need for them. For example, object-oriented abstractions such as classes provide the software engineer with encapsulation and a richer typing mechanism that uses data and function together, whilst inheritance allows the software engineer to represent reuse as well as providing a way to model more of the semantics of the domain, particularly semantics of methods. Blackboard architectures allow the software engineer to represent a particular pattern of control, that of changes in data triggering behaviour, whilst Prolog allows software engineers (amongst others) to represent and manipulate logic-based objects. This rich set of abstractions exist because they are used, because survival of the fittest allows them to remain. If they weren't used, they would not survive. This chapter has introduced and described a set of software entities (or abstractions) that hopefully complements these existing abstractions, whilst also providing improved evolveability.

This chapter has described a set of software entities, or abstractions, whose evolveability will be discussed in chapters 6, 7 and 8. The interfaces of these software entities has also been described, with emphasis on how these interfaces differ from interfaces of abstractions in existing software languages, models and architectures, and how these arguably improved interfaces help in identifying the effects of change on dependent software entities. This topic will be returned to in chapters 6, 7 and 8, which relate the ideas of software entity evolution spaces to their effects on the interface of a software entity.

Chapter 6

Data Evolution and Evolveability

1 Introduction

Data and data structures form a major part of any software system and are prone to change. These changes are triggered by:

- New requirements e.g. a user wanting to sort different types of data;
- Changes in other data structures e.g. a change in a super-class can affect any sub-classes or classes that reference the changing super-class.

Existing software systems depend on often complex data structures consisting of pointers, records and objects. These constructs allow many powerful data structures to be created, with the disadvantage that these data structures are sometimes difficult to comprehend and change without knowing what effects these changes will have on the consistency of the software system's data structures as a whole.

Many data modelling techniques adopt a domain-oriented approach to data modelling in order to improve the cohesion of the models, with the assumption that the inherent cohesion of domains will mean that changes to the data will be localised. However, the problems with any domain data model are that of:

- Domain definition: what defines a domain;
- Domain stability: how stable is the domain;
- Domain model stability: how stable is the domain model and how well does it model the domain.

In the 2D graph domain, for example, the data entities "node" and "edge" define the domain (the domain isn't defined in terms of its services because these may change). On the other hand, the sort domain isn't described in terms of data but in terms of a constraint on the input-output relation of its services i.e. all sort algorithms produce output that is sorted. Other domains are more difficult to define. Take the routing domain as an example. This domain consists of routing table data structures, routing data types such as routing ids and routing algorithms. Hence, the routing domain could be defined in terms of both routing data types and routing algorithms. But should this definition include all routing algorithms? A definition based on this would be incomplete with respect to any future routing algorithms. An alternative may be to base the definition on a generative definition which describes routing algorithms by their properties. One such property, as described above for the sort domain, is that of the input-output relation of an algorithm. This, however, is difficult to do for routing algorithms. Another approach is to define a domain in terms of concepts and terminology in the domain; for the routing example, these concepts would include those described above i.e. routing tables and algorithms (or services) which manipulate routing tables. In short, the existence of so many ways of defining a domain

makes it difficult to formulate a clear definition of "what is a domain". Hence, it is difficult to determine the attributes that characterise a domain because different domains are characterised by different attributes.

Most domains work with data that is in a particular format e.g. data in the graph domain consists of nodes and edges which have particular relationships with each other. However, there is a need to convert this data for use by other domains and services. This means that there must be a way for data to be mapped between domains, and this is another potential source of evolution. For example, instead of edges in the graph domain being mapped to lines, they may be mapped to arrows. This chapter describes a form of modularisation or abstraction which allows such information to be expressed, using DEMs. This form of abstraction is the DEM mapping.

Another form of abstraction is that created by data access modularity, in which access to data possesses its own abstraction. This is advantageous because data access abstractions can then be changed without affecting other parts of the code, unlike current software models in which data access code is intermingled with other code. There are three types of data access abstraction provided in the SEvEn framework:

- Data access services (see chapter 7 section 3.2.1.1);
- DEM paths, and;
- DIM paths (see sections 3.7and 3.8).

This chapter describes existing data modelling technology, its advantages and shortcomings. A data model based on graphs, consisting of DEMs and DIMs, is then described and compared with these existing techniques, followed by a discussion of the evolveability of data expressed in terms of this data model. The concept of an evolution space is used to describe the set of types of evolution that can occur to DEMs, DIMs and DEM mappings, how these types of change affect the interfaces of the software entities concerned, and how changes in these interfaces affect any dependent software entities. The inherent dependence of data on other data and the lack of dependence of data on functional aspects of software (the services), means that the analysis and improvement of data adaptability discussed in this chapter mostly concentrates on data adaptability issues with respect to evolution of data.

2 Existing Data Modelling Techniques

2.1 The Relational Model

Developed by Codd in the late 1970's, the relational model structures data using relations (or tables), which group together related attributes [Codd70a, Codd82a]. Relationships between tables are expressed through the use of shared attributes. The main problem with the relational model is implicit data typing. In addition, logical relationships (between tables) are second-class citizens, represented as shared attributes across tables. This makes it difficult to manipulate relationships directly.

2.2 Object-Oriented Models

Object-oriented data models consist of classes which encapsulate state (or data) and methods (functions which use and update the state) [Booch91a, Jacobson92a, Rumbaugh91a]. Classes are related by two types of relationship:

- Reference relationships, and;
- Inheritance relationships.

In a reference relationship with class A the parent and class B the child, B is given access to the methods and data of A (depending on the object-oriented model employed, this access could be only public methods and data, or may be extended to other methods and data). An inheritance relationship with class A the parent and class B the child allows class B access to the methods and data of class A. The differences between the two types of relationship are mainly to do with the behaviour of the relationship with respect to which methods are called. For example, polymorphism, when coupled with inheritance, allows the software to base which method is called on the type of the class. Inheritance can be expressed in terms of reference relationships with the appropriate addition of extra functionality to deal with polymorphism, for example. However, inheritance is a useful semantic construct that provides a different abstraction in addition to the reference relationship for modelling purposes.

The advantages of object-oriented models for software evolution have often been extolled, without much evidence to show their effectiveness in this area. Object-oriented models may improve the encapsulation and cohesion characteristics of software, but these in themselves can't prevent the software engineer from choosing inappropriate abstractions that will break when new requirements are introduced. Changes that break interfaces will always pose problems for software evolution and since software inherently has interfaces no matter which model is used, breaking interfaces will always occur. The question to be asked is whether object-oriented models have fewer interfaces and whether these interfaces are less likely to break. Inheritance, for example, introduces an interface between classes and the classes which inherit from them (their sub-classes). Hence, changes in a class may break the super/sub-class interface. Whether or not they will depends on the impact of the changes which, in turn, depends on a number of factors:

- The stability of the model with respect to what it is modelling. An unstable real-world application domain or a lack of understanding of the real-world application domain will result in a model that continually has to change;
- Whether or not the design of the model limits the scope of ripple effects. This is influenced by the stability of the abstractions in the model i.e. whether or not evolution will change the abstractions and the relationships between abstractions, or result in changes within abstractions which are easier to contain. Inter-class relationships encapsulate assumptions which may be broken by changes. For example, the removal of a data member from the state of a super-class may affect all sub-classes of this class.

The extra types of abstraction that object-oriented modelling introduces, such as the class and inheritance relationship abstractions, provides the software engineer with a richer modelling framework, and allows him to constrain software in a number of ways by, for example, being able to specify the type of class that should be used in a particular situation. For instance, a filter program can be designed with a single thread of control which calls input, filter and output

functional components (represented, perhaps, as class methods) in sequence. By sub-classing these three types of functional component, different types of filter program can be created. The use of classes has therefore provided constraints on the software such that any functional components used must sub-class one of the existing input, filter or output classes. This can be useful. It can also be problematic when new requirements break the constraints imposed by the software engineer who has utilised the object-oriented abstractions provided. In particular, the flexibility of the software has been sacrificed for improved design characteristics such as improved semantic constraints such as the one that has just been described.

2.3 Entity-Relationship Models

The entity-relationship model was originally proposed as a way of unifying the network and relational data models, and permit the expression of a problem using entities, relationships and attributes [Chen76a]. The relationships exist between entities, which may possess attributes. ERA models are a good *generic* model for data modelling. They allow relationships to be represented as first-class entities within the model, but don't provide a way for modelling the relationships between the data and functional elements of the software, which is required in software evolution in order to determine how changes in a software entity affect other software entities.

2.4 EXPRESS

EXPRESS, EXPRESS-C and EXPRESS-M are an object-flavoured family of languages used for specifying information models [Schenck94a]. The main language is EXPRESS in which information models consist of data entities, rules (or constraints), basic unbounded data types (for example, the set of all integers), and schemas. EXPRESS-C extends EXPRESS by allowing services to be added to data entities, pushing EXPRESS more towards an object-based modelling language. EXPRESS-M is a schema mapping language, developed as part of a Ph.D. project, which consists of constructs for describing how instance data in one information model can be converted or translated into instance data in another information model [Cimio95a]. EXPRESS-M is related to DEM mapping software entities, which are described in section 3.9.

3 Data-Based Software Entities

The data models described in the previous section provide a set of entities for modelling data. Their primary disadvantage is their lack of theory and support for software evolution. This chapter attempts to improve upon this state of affairs by developing a new data model with improved support for software evolution. To this end, this section describes the data model adopted throughout this thesis, consisting of a data entity model (DEM) and a data instance model (DIM). It is based on the ERA model (see section 2.3) but, unlike the ERA model, provides support for software evolution, improved adaptability and localisation of evolution, along with improved modelling with respect to dependencies on other software entities in a software system.

3.1 DEMs (Data Entity Models) and DIMs (Data Instance Models)

A major problem with the "niche" data models (they are niche data models because they are designed to model particular types of data) described in the previous section is the specificity of the models: the models are designed to model specific information in a specific way. For this reason they are inflexible and difficult to change to model the types of information for which they were not originally designed. This presents no problems if changes in data requirements don't fall outside the boundary provided by the niche data model, but will inevitably cause problems if changes in requirements need to model data which is incompatible with the existing model. For example, a tree-based data model is no good for modelling graph or network-based data.

There are two approaches to overcoming this limitation:

1. Produce a generic data model which allows all other data models to be expressed in it. In addition, provide a set of transformations from these "niche" data models to the generic model so that transformations between "niche" data models can be "brokered" through the generic data model;
2. Represent all data using a generic data model, designed in such a way that all potential data modelling requirements can be met by the model.

DEMs and DIMs aim fulfil the role of this generic data model. In this thesis they are used to model data in the sense of the second approach for simplicity. However, the first approach probably offers better advantages over the second approach because:

- It provides the advantages of a brokered data mapping. This means that, for N "niche" data models, there need only be N data mappings rather than N(N – 1), because all mappings go through the generic data model (see Figure 1);
- It allows data to be represented and manipulated using the best model for the task – the niche models.

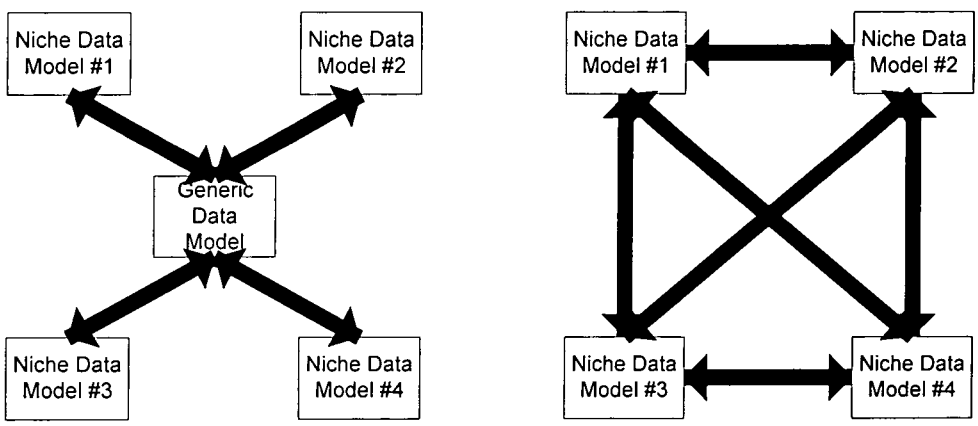


Figure 1 - Data Model Types and Data Mappings

The approach of this thesis is to use a data model that satisfies the following criteria:

- Not dependent upon the domain which it is modelling;

- As generic (or abstract) as possible by limiting the constraints imposed by the data model and thereby improving the flexibility of the data model with respect to changes in data modelling requirements.

The first point is necessary so that changes in the domain don't break the model. For example, modelling the rules in a tax domain as a fixed array would invalidate this rule because the addition of new rules would break the model. However, modelling the rules within a production rules model would allow new rules to be added without breaking the model. The second point is important in the sense that it allows many different domains to be modelled. For example, the production rules model is sufficient for a number of modelling tasks, but would fail to capture the essence of a lot of other domains such as a graph domain.

The idea is to raise the level of abstraction of changes made to domain data above that of predicting whether and how a data structure might change. For example, in the email application domain, a software designer may predict particular changes in the domain data itself, such as the addition of a new header field. Depending upon the particular data structures chosen to represent data in this application domain, this may or may not be an easy task. It is one of the hypotheses of this thesis that raising the level of abstraction higher to that of a DEM (Data Entity Model), offers benefits in this regard because a DEM (directed-graph-based data model) is the most general data model possible. A graph can be used to represent any data structure, including graphs themselves. For example, the addition of a new header field can be accomplished simply by adding a new data entity[1] to the appropriate place in the model, the model itself ensuring through existing constraints, that this new addition is valid for the model. This represents a different approach to modelling application domain data, and is similar in some respects to that of the shift from imperative to declarative languages that occurred in the late 1980's, which moved the programming task away from the "how" to the "what". Similarly, the shift from traditional data structure models to DEMs implies moving away from such concepts as arrays, records and lists to data entities and relationships[2]. The abstract nature of the model allows the software engineer to describe what he wants to change (by adding a new data entity or relationship to the model) and allowing the (adaptation space) constraints to determine the effects of these changes.

By using DEMs, the software engineer can capture the main concepts of a domain and design a meta model that describes how the domain concepts relate to each other, and how DEMs as a whole relate to each other. The latter provides a way to link DEMs via specialisation relationships so that, for example, the user interface domain can be specialised for particular domains and the specialisation information recorded. Take, for example, the user interface domain. User interfaces are quite volatile in the early stages of software development, as users change the way they want

---

[1] The terms data entity and DEM are subtly different. A DEM refers to a set of data entities related through *HasA* and *IsA* relationships. There is a similar distinction for the terms data instance and DIM; a DIM is a set of data instances related through *HasA*^{I} relationships.

[2] This shift may or may not ease maintenance in a particular situation - this will depend on the exact design and implementation of the DEM taken. It is interesting to note that the supposed eased maintenance benefits of declarative i.e. 4GL programming languages have not been accrued, possibly through bad design or due to an emphasis on ease of development and a lack of emphasis on easing evolution.

their software to look. However, even though the look of the software may change, the underlying concepts such as list boxes, menus, text boxes etc. are fairly static once modelled. Hence, changes to the domain generally don't affect the existing model but extend it with more concepts. Evolution in this case implies a change in the number of instances of the concepts that are used i.e. a change in the DIM (Data Instance Model) rather than the DEM. However, in other cases, the user interface domain itself (i.e. the DEM) may evolve to include more concepts, such as radio buttons and check boxes. Another example is the telecommunications domain. The data in this domain basically consists of a caller telephone number and a callee telephone number, which are used by the services provided by a POTS (Plain Old Telephone System) telephone switch. New services such as call minder; call waiting and call redirection are added periodically to the basic switch. These new services require extra data, in addition to the basic caller and callee information. One could view this new data as being part of the existing POTS domain, which would then result in a domain that is prone to change. However, if one views the telecommunications domain as consisting of a non-empty set of sub-domains, then evolution would occur by the addition of new domains rather than by changes to existing domains. For example the DEM for a new call minder service could be *based on* (or expressed in terms of) the DEM for POTS, rather than having a monolithic domain consisting of POTS, call minder and call redirection. This results in domains that are static and which don't need to evolve, because data evolution occurs at the inter-domain level by the addition of new domains.

DEMs are an implementation of the most general data model possible, a directed graph. They are a specialised form of software entity model in which all software entities in the DEM are data entities and relationships are *IsA* and *HasA* relationships (similarly, a DIM is a software entity model in which all software entities are data instances and relationships are $HasA^I$ relationships). They can be used to represent any possible data structure, from primitive data types such as strings and integers, to complex data structures such as linked lists and graphs. Their structure consists of data entities (primitive and non-primitive) linked by binary relationships (consisting of a parent data entity and a child data entity), which are of two types:

- *HasA* relationships, which include a cardinality, C, and signify that the parent data entity has C child data entities;
- *IsA* relationships, which signify that the parent data entity is a sub-type of the child entity. The semantics of this relationship are object-oriented inheritance semantics.

It should be noted that, because of the way DEMs model data, there is a subtle interaction between *HasA* and *IsA* which means that 1 implies 2:

1. A *HasA* B;
2. B *IsA* C and A *HasA* C.

Hence, data that falls into the second category can be modelled using the style of the first category. The reason for this is the existence in DEMs of an "implicit *IsA*".

A DIM is a directed graph that consists of data instances (which are instances of data entities in a DEM), related only by *HasA* relationships. A DIM is an instance of a particular DEM, and must be valid with respect to the DEM. This validity

is determined by the *InstanceOf* relationship which requires that the parent (data instance) of the relationship must be a valid data instance of the child (data entity) of the relationship, applied to the whole DIM. A data instance is a valid instance of a data entity if the child sub-model of the parent data instance is valid with respect to the child sub-model of the parent data entity. The data instance sub-model is valid with respect to the data entity sub-model if each instance sub-model relationship and child instance is valid with respect to the data entity sub-model (including inheritance provided by any *IsA* relationships).

The following observation is important. If one partitions a domain into data and function, then all data within that domain can be described in terms of a set of basic types, such as String, Integer, Float, Character, Date, GIF, etc. It is the way that these basic (or primitive) types are structured and used by the services of the domain that characterises the data as belonging to a particular domain. For example, the structure of data in the Relational Database domain is very different from that in the Graph domain and it is used differently by the services in that domain, even though both domains use the same primitive types. Hence, domain data is uniquely identified as belonging to a particular domain by the structure of the data, the types used and the services. A DEM is an entity model in which the entities are constrained to be data entities. The leaves of the DEM graph model are primitive data entities whilst the other, non-primitive, data entities along with the relationships provide structure to the primitive data entities.

	Relationship	Parent	Child
1	*HasA*:<Cardinality>	NPE	PE
	PartOf:<Cardinality>	PE	NPE
2	*IsA*	PE	PE
	IsAInv	PE	PE

PE = the set of primitive data entities

NPE = the set of non-primitive data entities

$E = PE \cup NPE$

Table 1 - DEM Relationship Types

HasA relationships possess a cardinality (see Table 2).

Cardinality	Constraints	Description
N		Any number of children.
=x	x ∈ nat	Exactly x children.
<=x	x ∈ nat	Any number of children up to and including x.

Table 2 – *HasA* Cardinalities

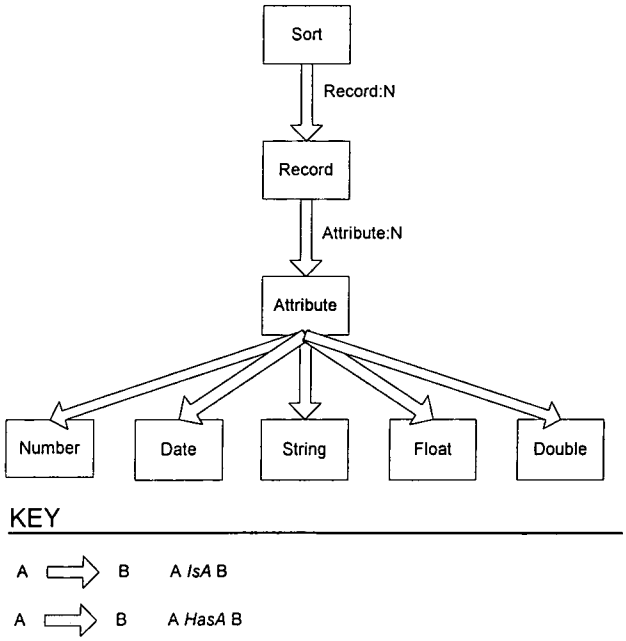


Figure 2 - DEM_{Sort}

Data entities are very much like objects; they encapsulate data (state) and services act upon that data, even though DEMs don't provide a way to encapsulate these two aspects together because cohesion can be a form of inflexibility when evolution needs to occur. The primary characteristic of flexibility is that of lack of constraints; if there are no constraints then there is no "inertia" to make changes more difficult to make. Of course, there is no such thing as a completely unconstrained design, which means that particular structures and relationships are necessary. These include:

- Relationships between functional software entities;
- Relationships between functional and data software entities.

The exact form of these relationships is not important, as long as they exist because they are an inherent part of the design of a software system. Any other relationships and structures, such as objects, aren't necessary but are meant to improve some characteristic of the software. In doing this, they impose more constraints to change and hence decrease the flexibility of the software. The design of the DEM model, and indeed of the whole SEvEn model, attempts to limit the number of constraints in software in order to improve the overall flexibility of the software.

As an example, the sort domain may contain a DEM of the form shown in Figure 2. The services BubbleSort and CompareTo must be expressed in terms of the DEM; that is both services must use the data entities and relationships in the DEM to perform their task. This, however, introduces an implicit dependency between the services and the DEM such that if the DEM changes then the services would inevitably have to change. The thesis will return to this issue in chapter 7 when it discusses service adaptability with respect to DEM evolution.

```
Sort.BubbleSort (DIM Sort) {
    for (int i = NumElements (Sort.[Sort 0,Record]) - 1; i > 1; i--) {
        for (int j = 0; j < i - 1; j++) {
            if (Compare (Sort.[Record, j], Sort.[Record, j + 1]) > 0) {
                Swap (Sort.[Record, j], Sort.[Record, j + 1]);
            }
        }
    }
}
```

Figure 3 - Sort.BubbleSort

```
RDB.Select (DIM RDB, DIM Parameters) {
        Boolean matches;
        DIM NewRDB = cloneDIM (RDB);
        for (int i = 0; i < NumElements (RDB.[RDB 0,Table 0,Record]); i++) {
            matches = true;
            for (int j = 0; j < NumElements (Parameters.[BaseEntityName]; j++) {
                InstanceValue IV1 = RDB.[RDB 0,Table 0,Record  i,Attr
                    Parameters.[Parameters 0,Record j,AttributeNumber 0]];
                InstanceValue IV2 = Parameters.[Parameters 0,Record j,AttributeValue 0];
                if (Compare (IV1, IV2) != 0) {
                    matches = false;
                }
            }
            If (matches) {
                % Add this record to new RDB
                NewRDB.Add (NewRDB.(RDB 0,Table 0,Record -1), RDB.[RDB 0,Table 0,Record i]);
            }
        }
}
```

Figure 4 - RDB.Select Service[3]

In Figure 3 and Figure 4:

- Boldface text represents services;
- Italicised text represents DIM-usage using DIM paths (see section 3.8);
- Underlined text refers to DEM and DIM data types;

[3] RDB is an acronym for Relational Database.

- % is a comment.

The RDB.Select service requires two parameters:

1. The attribute to select;
2. The value of the attribute to select.

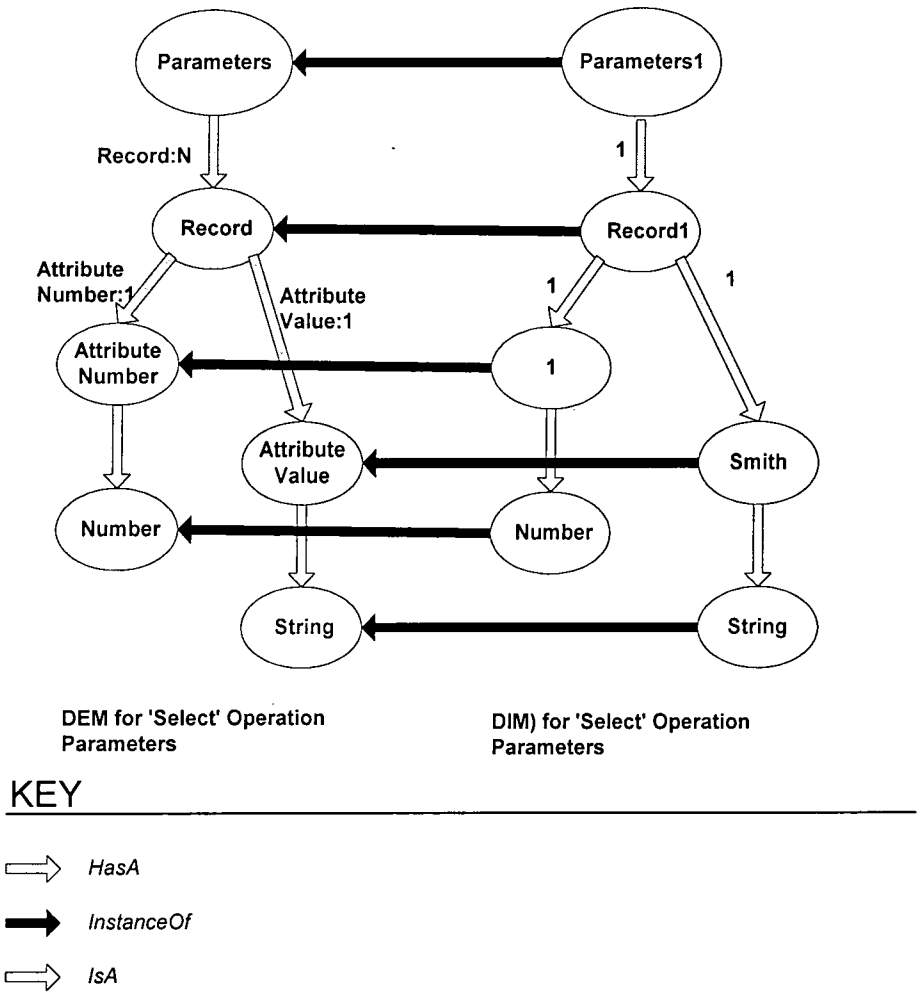


Figure 5 – The Relationship Between a DEM and a DIM

In keeping with the spirit of the modelling approach taken in SEvEn, parameters should also be in the form of a DEM. This also aids in validating parameters i.e. ensuring that they are of the correct "type". For example, the parameters to the RDB.Select service (see Figure 4) could be represented as the DEM shown in Figure 5.

DEMs allow data validation because they specify the structure of the domain, thus allowing DIMs to be validated against DEMs. They also provide a shared (domain) vocabulary over which FSEs can co-ordinate and potentially negotiate data conversions using DEM mappings (a potential source of evolution in software).
The relationship between a DIM and the DEM of which it is an instance, is shown in Figure 5.

3.2 Using DEMs and DIMs to Model Common Data Structures

In order for DEMs and DIMs to be useful in software, it must be possible to model various common data structures utilised in software. Evidence of their wide applicability is shown in this section by examples of modelling common data structures in terms of DEMs and DIMs.

3.2.1 Enumeration Types

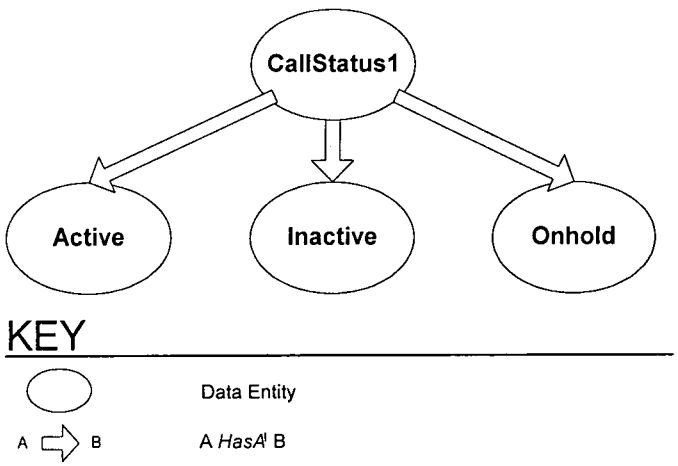


Figure 6 - A DIM Used to Implement an Enumeration Type 'CallStatus'

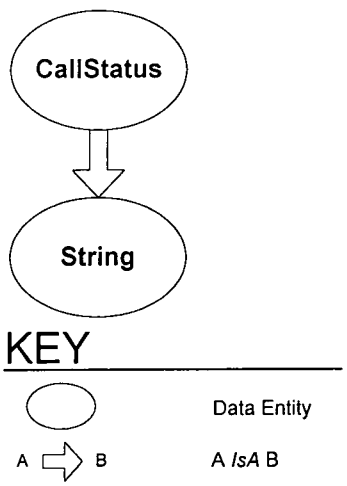


Figure 7 - $DEM_{CallStatus}$

Take, for example, an enumeration type called CallStatus for recording the status of a telephone call. This could be specified using set notation as: {Active, Inactive, Onhold} and represented in a computer as an integer sub-range, the software developer providing the range checking code if it isn't built into the implementation language directly. An enumeration type can also be represented using a combination of a DIM that holds the important enumeration values and a DEM that specifies the structure of the DIM.

3.2.2 Sub-range Types

Take, for example, a sub-range type used to represent the age of the customers of a telecommunications company. A sub-range type represents a range of values of a base type in a succinct way, perhaps because the range is too large to explicitly represent each member. In the age type example, the base type could either be an integer to represent whole ages or a floating point number to represent fractional ages. This could be represented using the DIM and DEM in Figure 8 and Figure 9.

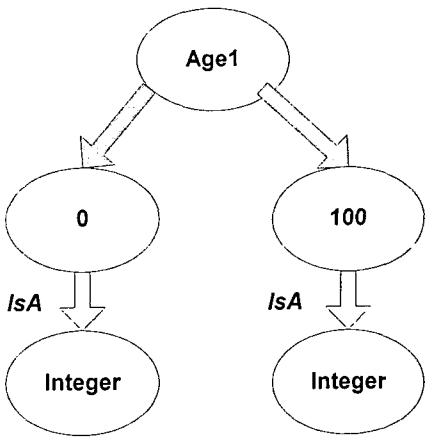


Figure 8 - DIM Used to Represent Age Sub-range Type

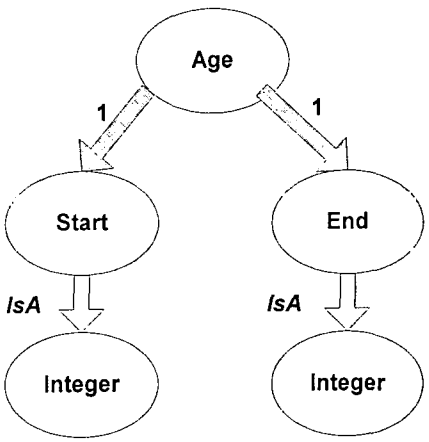


Figure 9 - The DEM for the Age Subrange Type DIM

3.2.3 Hashtables/Association Lists

A hashtable/association list can be modelled in terms of a DEM as shown in Figure 10. In addition, there needs to be a way to access data entities for a particular requirement. In this case, the service shown in Figure 11 will accomplish this for a given key.

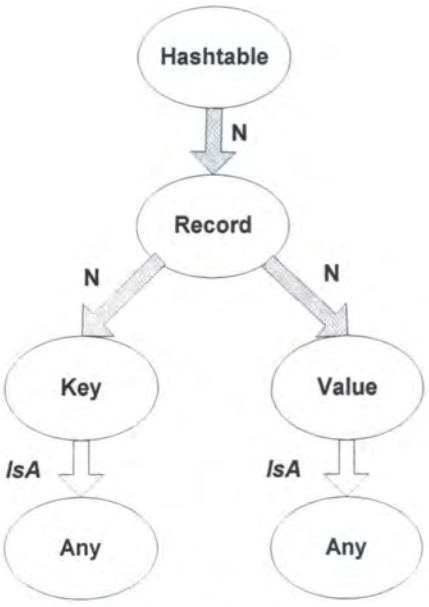


Figure 10 - Hashtable DEM

```
DEM HashValue (DEM Hashtable, Any Key) {
    return Children (Parent (Key))
}
```

Figure 11 - Constraint for Hashtable

This service takes as parameters the DEM representing the hashtable and the required key, and returns all siblings of the key in the hashtable.

3.3 Data Services

Data services are a set of domain-independent, mutually-exclusive services (see Table 3) which can be performed on DEMs and provide primitive DEM-manipulation operations. They provide a base for building domain-dependent services.

Data Service	Formal Parameters	Description
Swap	DIM Data, DIMPath Left, DIMPath Right	The two data instances pointed to by the two DIMPaths "Left" and "Right" in the DIM "Data" are swapped.
Compare	DIM Data, DIMPath Left, DIMPath Right	The two data instances pointed to by the two DIMPaths "Left" and "Right" in the DIM "Data" are compared. -1 is returned if "Left" is less than "Right". 1 is returned if "Left" is greater than "Right". 0 is returned if "Left" and "Right" are equal.
Get	DIM Data, DIMPath P	The value of the data instance pointed to by "P" in the DIM "Data" is returned.

Set	DIM Data, DIMPath P, Data Instance I	The value of the data instance pointed to by "P" in the DIM "Data" is set to the value "I".
Parent	DataEntity DE	Returns the parent data entity of "DE", reachable through a *HasA* relationship.
Children	DataEntity DE	Returns the children data entities of "DE", if any, reachable through *HasA* relationships.
Siblings	DataEntity DE	Returns the siblings of "DE", reachable through *HasA* relationships.

Table 3 – Example Data Services

3.4 DEM and DIM Advantages and Disadvantages

A major advantage of DEMs is their ability to allow sharing of heterogeneous data, as Figure 12 shows. The genericity of DEMs allows data structures in different languages to be represented using one, common representation. This cuts down on the number of converters required (for N data structures, N converters are required instead of N(N – 1) as discussed in section 3.1).

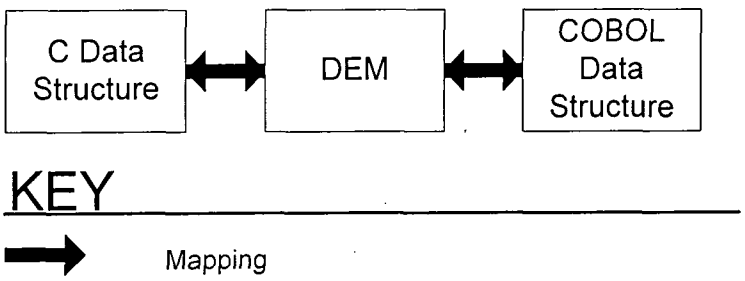


Figure 12 - DEM as a Common Data Structure

An important issue for DEMs and DIMs is that of scalability. Will large and complex domain representations be representable using DEMs? Large and complex domains are generally made up of smaller, simpler domains which can be represented using DEMs which are quite small, easy to navigate, and easy to convert to other DEMs.

DEMs provide for data validation because they specify the structure and constraints of the domain data and prescribe the structure of any data that purports to be of a particular representation or domain. However, a domain can't be fully described in terms of its domain data, which is represented by a DEM. There is more to a domain than simply the types of data entities in that domain and the relationships between these data entities. Services also play an important part in defining the semantics of the domain. Using just the DEM as semantics for the domain would be naive, since two DEMs with the same structure[4] don't necessarily model the same domain. Other factors come into play, such as the semantics

[4] Two DEMs have the same structure when they have the same data entities and relationships i.e. when the DEMs can't be differentiated.

of the primitive data entities used in the domain[5], and the services that describe how the data is used. For example, consider the Sort and Output DEMs (Figure 2 and Figure 17 respectively), which both have the same structure but which are trying to model different domains. Requirements engineers face similar problems, which they attempt to solve by using viewpoints, i.e. a model is useful only when used in a particular context or in a particular way. This idea can be applied to the sort and output example by viewing FSEs as stakeholders in DEMs, which provide different viewpoints on what is essentially the same data which is used in different ways. In summary, it is the way the data is used that helps to categorise it as being in a particular domain.

Unless a formal model of the semantics of the DEM exists, the users must perform the mapping between two domains since it is they who know the semantics of the data entities and therefore the valid semantic mappings between data entities. It is also the users who need to decide, in situations where more than one valid mapping between two domains exists, which mapping to choose. For example, in the Graph $\Rightarrow$ Output mapping should nodes be represented as circles or squares?

In summary, DEMs provide a way of modelling data that uses the most general mechanism – a directed graph that can be used to represent any other data structure – and provide a shared vocabulary over which different FSEs can co-ordinate, negotiate and share information. In addition, some domains have in-built data structures that may change e.g. the income tax domain has tax rules. The DEM can be used to represent these domain data structures in a way that eases changes made to the domain data, and provides a common representation mechanism for converting data and allowing such data conversions to be modelled as a separate software entity. Mappings between DEMs then provide a way for evolution between DEMs to occur. Another advantage of DEMs is that they allow domain concepts to be captured explicitly and reasoned about and manipulated.

Data structures in software are commonly unconstrained (or fairly flexible) and complicated, especially when pointers are used. DEMs offer an advantage in that they explicitly model the domain data in a way that is easy to change, supported by the relevant mechanisms. They also offer a common way of specifying data structures i.e. in the form of data entities and relationships, rather than using different mechanisms for different types of data structure. This allows a common data mapping approach to be used for data conversions.

3.5 DEM Interfaces

It is important to identify what comprises the interface of a DEM, so that changes in a DEM can be related to changes in the DEM interface, and thereby linked to ripple effect types on dependants of the DEM.

The interface of a DEM comprises:

---

[5] For example, in the telecommunications domain, the status of a telephone call can take on two states: active (both parties are connected together) or on hold (one party has been temporarily disconnected). These states can be modelled as a subtype of the type INTEGER with two values, say 0 and 1. Attaching semantics to this subtype for this particular domain would involve making explicit the fact that 0 represents ACTIVE and 1 represents ON HOLD.

- The structure of the DEM;
- The semantics of the DEM.

Hence, any software entity that depends on (is related to) a DEM depends on the structure and the semantics of the DEM. It is important to choose the type of dependency of clients on DEMs so that adaptability is maximised. A dependency with an in-built assumption of structure leads to lack of adaptability, as Lieberherr et al have found through the Demeter project. This is an example of white box abstraction, as shown in Figure 13. X is a DIM, an instance of the DEM Y which has a child data entity, Z. The removal of data entity Z from Y invalidates the DIM X, which is now invalid with respect to Y.

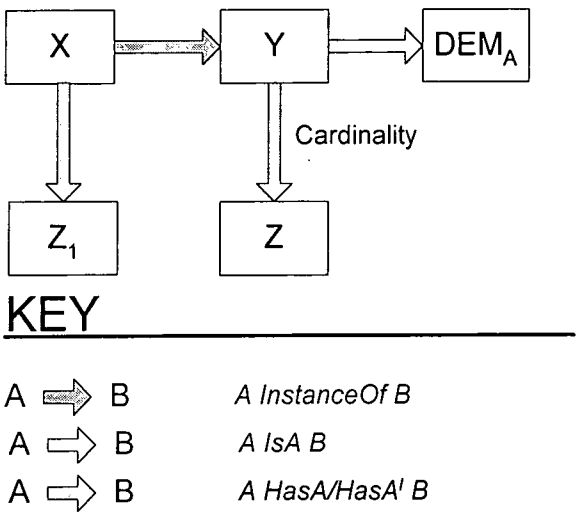


Figure 13 - Dependence of Instances on DEM Structure

However, there is also a dependence of clients on data semantics, which poses problems for adaptability since it breaks the requirements of the client.

In order to improve the adaptability of software entities with respect to DEMs requires the use of a different type of interface which hides the structure of the DEM in some way. This is discussed in section 3.7.

3.6 DIM Interfaces

Just as it is important to determine the interface of a DEM, it is also important to determine the interface of a DIM. A DIM consists of:

- Data instances, and;
- Relationships;

and the main items of interest to client software entities such as service instances are the data instances. Hence, the interface of a DIM consists of the data instances which comprise part of the DIM. The choice of form of this interface is

important in improving the adaptability of clients. Section 3.8 discusses DIM Paths as a form of interface for DIMs which allows improved adaptability of clients with respect to changes in the DIM.

3.7 DEM Path Software Entities

There needs to be a method of uniquely identifying a particular data entity. There are two ways of accomplishing this:

1. A broker-based approach in which a unique identifier, a data entity descriptor, is assigned to each data entity which is used by a client to access the data entity and which is used to provide a mapping to the actual data entity;
2. A more distributed approach in which access to data entities is by the use of DEM paths, which uniquely and explicitly identify a data entity in a particular DEM.

Both approaches can utilise the notion of DEM paths. Option 1 can map data entity descriptors to DEM paths. Option 2 uses DEM paths explicitly.

In DEM paths, data entities are represented using data entity descriptors (DEDs), which are unique only with respect to the DEM of which they are a member. A DEM path which doesn't encapsulate hard relationships between data entities as in most existing software models is of the form:

DEM Path ::= <DED>

i.e. a sequence of data entity descriptors beginning with the base data entity and ending at either a non-leaf or leaf data entity. The data entity to which the DEM path as a whole refers, is determined by tracing through the DEM from the base data entity through to the last data entity in the DEM path. The primary advantage of this type of interface to data entities is that data entities can be moved around within a DEM as a result of particular changes in the structure of the DEM, without affecting the DEM Path. Any software entities requiring access to such a data entity will then not be affected by such changes, because the DEM Path is adaptable with respect to such changes.

3.8 DIM Path Software Entities

The existence of DIM path software entities has a similar justification as that of DEM paths. Data instances are represented using both DEDs and data instance descriptors (DIDs). DIDs are only unique with respect to the DIM of which they are members. Hence, data instances are unique only with respect to their DIM and, by implication, the DEM of which the DIM is an instance. DIM paths are also used in services. This aspect of their use is described in chapter 7, when service adaptability with respect to changes in data is discussed.

A DIM path is of the form:

DIM Path ::= [<DIMName>] [X] <Y>

X ::= DED DID

Y ::= DED I | <parent>

where:

- <...> indicates a sequence;
- [] indicates an optional element;
- I is a natural number between 1 and NumInstances (InstancePath) inclusive that signifies the Ith data instance in the DIM of the last DED in the DIM path;
- DED is a data entity descriptor, a unique data entity identifier;
- DID is a data instance descriptor, a unique data instance identifier within the context of a particular entity;
- NumInstances (InstancePath) returns the number of instances of the last entity in the DIM path;
- The <parent> tag signifies a move up the DIM hierarchy to the parent data instance. This is a useful construct for allowing flexible movement through a DIM. It can be used, for example, for accessing siblings of a particular data instance.

The elements of a DIM path are called DIM path components (DPCs). A DIM path uniquely identifies a specific data instance in a DIM that starts with an instance signified by the start DPC and ends with an instance of another entity, which is reachable from the start data instance. Square brackets around a DIM path refer to the actual value of the data instance concerned whereas curved brackets refer to the primitive instance itself. In analogy to pointers in the C language, (DIMPath) is a pointer to the data and [DIMPath] is a de-reference of the data.

DIM paths that are going to be de-referenced must end at a primitive data entity so that the DIM path as a whole can be treated as a pointer to a primitive data instance. DIM paths that don't end with a primitive data entity implicitly refer to more than one instance of that data entity, because any non-primitive data entity instance has more than one child instance. In this case, the instances obtained from the DIM path will be descendants of the first instance in the DIM path. If the DIM path begins with a non-base data entity, the instances obtained from the DIM path will be descendants of each instance of the start entity.

The primary advantage of DIM paths is similar to the advantage of DEM paths described in the previous section, namely that a DIM path provides adaptability with respect to changes in DIM structure.

3.9 DEM Mapping Software Entities

Domain data conversion is necessary in order to allow domain-oriented components to communicate or integrate their resources in order to perform inter-domain tasks. The integration of domains is accomplished through the use of the DEM. The conversion of domain data, for example between the 2D graph domain and the screen output domain, occurs by the transformation between primitive data entities in the separate DEMs or schema's, according to a set of rules or heuristics that indicate how the data in one domain can be transformed into data in another domain. If the Graph and Output domains are modelled as in Figure 15 and Figure 17, then typical rules in the Graph to Output domain transformation may be as shown in Figure 14.

DEM mappings can be represented using DEMs, just like any other data structure. The advantage of representing them this way is to provide a common representation for all data in a software system.

For each Graph.Edge instance, E, create an Output.Line instance, L, and:

- Map E.Node[1].X ⇒ L.X1;
- Map E.Node[1].Y ⇒ L.Y1;
- Map E.Node[2].X ⇒ L.X2;
- Map E.Node[2].Y ⇒ L.Y2;

For each Graph.Node instance, N, create an Output.Circle instance, C, and:

- Map N.X ⇒ C.X;
- Map N.Y ⇒ C.Y;
- Map '5' ⇒ C.Radius, where '5' indicates a constant.

Figure 14 - A DEM Mapping from the Graph Domain to the Figure Domain

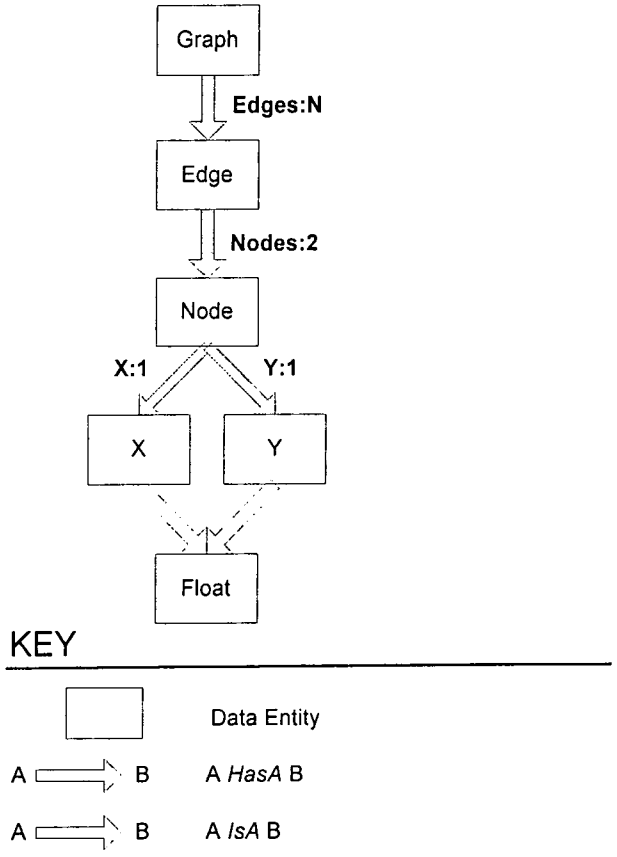


Figure 15 - $DEM_{2DGraph}$

Data mappings exist implicitly between functional elements in traditional software to convert data from one format into another format (see Figure 16). They are necessary so that functional elements can share data, when these functional elements are expressed in terms of different and heterogeneous data representations. The relationship between these different representations can be of two types:

- The relationship is syntactic. The two data representations are different "views" of the same semantic data. For example, two different views, or representations, of the same graph data;
- The relationship is semantic. The two data representations represent semantically different data. For example, one representation is for graph data and the other representation is for graphics output data.

DEM mappings can be used to represent both types of relationship. Purely syntactic DEM mappings are potential targets for some form of automated mapping (discussed further in section 3.9.5), which is possible because of the semantic equivalence of the two DEMs. DEM mappings are also prone to change when the services change. There is thus a dependency between DEM mappings and the services which use them.

Figure 16 - Data Converters and Services

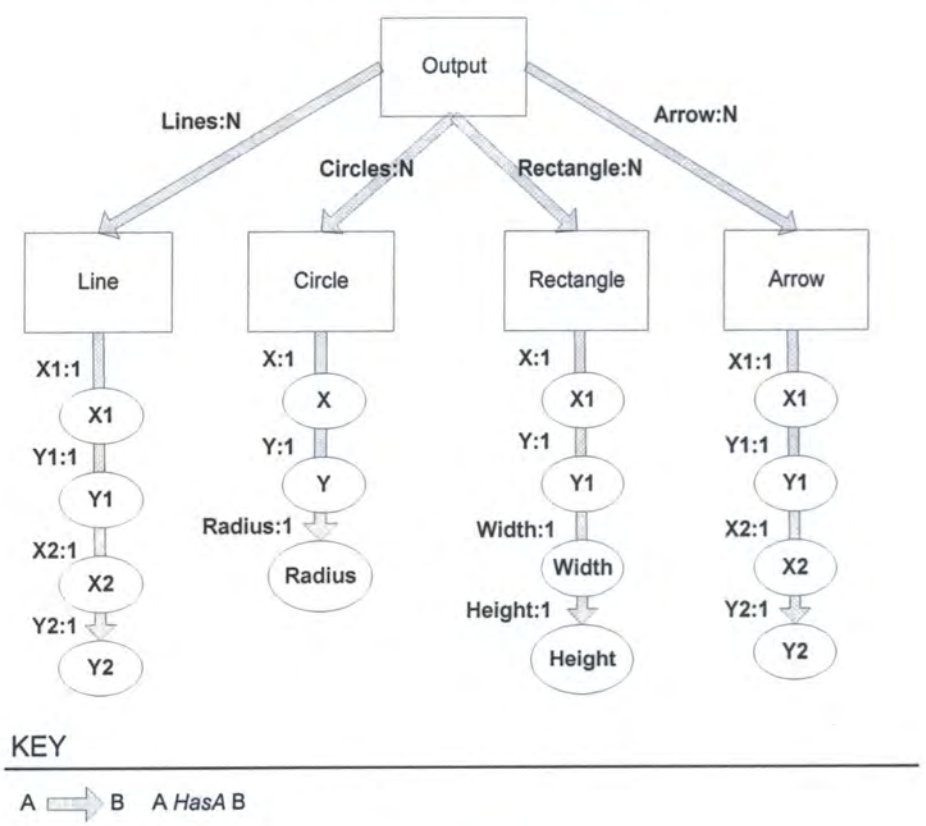


Figure 17 - DEM$_{Output}$

DEMs can be used to represent the transformation of data between different DEMs. The conversion between two DEMs is dependent on both DEMs, but doesn't belong to any one particular domain. It is also typically a semantic mechanism,

rather than syntactic, since it depends on the interpretation of the domain terms or data entities involved[6]. This conversion mechanism can be realised through the use of rules, each consisting of a pre-condition and post-condition that explicitly show how data from one domain can be represented in another domain. In SEvEn, software systems encapsulate domain data and domain services. It would not be appropriate to encapsulate conversion rules within individual software systems, since they may be global in nature, manifesting themselves in many different software systems' interactions. It is therefore the responsibility of a data mapping broker (a server FSE that is accessible through a well-defined interface by all other FSEs) to broker the use of data mappings.

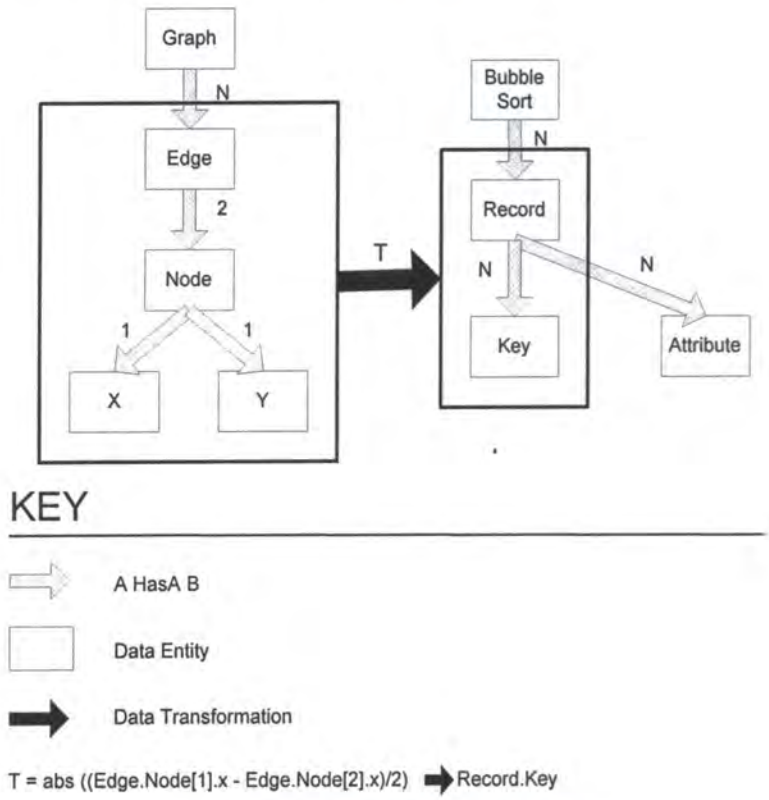


Figure 18 - Non-Trivial DEM Adaptation

The scope of transformations identified in the previous section excludes the possibility of the mapping shown in Figure 18, because they only allow the expression of structural transformations of the data. However, in addition to structure, the primitive data entities in a data model each possess a value which can also be part of the transformation. This is the case with the transformation shown in Figure 18, in which the value of Record.Key depends on the values of Node[1].x and Node[2].x. This observation brings us to a discussion of how transformations (as an expression of the evolution of data) can affect the data's dependants. There are two types of mapping relevant here:

- Changes in the structure;
- Changes in the values of primitive data entities.

[6] In some cases, conversions are syntactic; for example when the conversion is axiomatic (well known and universal) and when there are no choices for the conversion.

Hence, some DEM transformations can't be expressed entirely in terms of the adaptation operators employed by Hursch. Some transformations, like the one above, require the use of services, in this case subtraction and "abs". So, DEM transformations employ the use of both data evolution operators (identified in the previous section) and domain services written by the software engineer.

A DEM mapping specifies how to represent a particular domain's data in terms of another domain, much like the EXPRESS-M language permits the expression of data conversions in EXPRESS using both declarative and procedural information [Liebich95a]. The syntax of a DEM mapping is as shown in Figure 19. A DEM mapping describes how to map the children of a data entity. It is a recursive software entity because mapping a data entity results in mapping its children, which results in mapping their children, and so on.

```
DEM Mapping ::= {Entity Mapping}
Entity Mapping ::= <From Entity> <To Entity> <Child Mapping>+
Child Mapping ::= <From Path Instance> <From Path Instance>
From Path Instance ::= <Constant> | <PathA>
To Path Instance ::= <PathB>
PathA ::= <Entity Filter>*
PathB ::= <EntityI>*
Entity Filter ::= <Entity> 'N' | <EntityI>
EntityI ::= <Entity> <I>
<I> ::= [1..inf]
From Entity, To Entity, Entity ::= DATA ENTITY DESCRIPTOR
```

Figure 19 - The Format of a DEM Mapping

```
DEM Schema Mapping ::= <From Domain> <To Domain> <Entity Mapping>+
```

Figure 20 - The Format of a DEM Schema Mapping

A plus '+' indicates repeat one or more times, an asterisk '*' indicates repeat zero or more times. A vertical bar '|' indicates a disjunction and a quoted term such as 'N' indicates a constant. [1..inf] indicates the range of integers from 1 upwards. A DATA ENTITY DESCRIPTOR (DED) is an integer (greater than or equal to one) which uniquely identifies a data entity within a domain.

Informally, a data entity mapping specifies the mapping between the instances of a data entity in one domain to a set of instances of a data entity in another domain, by describing how the former data entity's children can be mapped. For example, Figure 14 shows how an edge in the graph domain can be mapped to a line in the figure domain by describing how the X and Y co-ordinates a node can be mapped to the X1, Y1, X2, Y2 co-ordinates of a line. Figure 19 shows that a constant can be mapped to a data entity instance. For example, there is no data in the graph domain that indicates what the values of circles in the figure domain should be. In this case, a constant value is mapped to the circle radius.

An <Entity Filter> consists of either a data entity followed by the specific data entity instance (indicated by an <I>) or by an 'N'. The former maps a specific data entity instance, whilst the latter maps all data entity instances for that particular data entity and is useful when all children instances of a particular data entity instance need to be mapped. For example, the DEM mapping for a sort DEM (see Figure 2) to an output DEM (see Figure 17) would consist of one data entity mapping from Sort.Record to Output.Record, consisting of two child mappings. The first child mapping would be from Sort.Key to Output.Key and would take the form Sort.Key N $\Rightarrow$ Output.Key 1, whilst the second child mapping would be from Sort.Attribute to Output.Attribute and would take the form Sort.Attribute N $\Rightarrow$ Output.Attribute 1. In other words, map the set of Sort.Key data entity instances $\{SKI_1, SKI_2, \ldots, SKI_n\}$ to the set of Output.Key data entity instances $\{OKI_1, OKI_2, \ldots, OKI_n\}$, starting at position one and preserving their ordering. The child mapping Sort.Key N $\Rightarrow$ Output.Key 2 would result in mapping the set of Sort.Key data entity instances to the set of Output.Key data entity instances, starting at position two.

It should be noted that both the from and to DIM paths must terminate on a primitive data entity, since it is the values of the primitive data entity instances which are being mapped. The rules ensure the preservation of the target DEM's structure.

3.9.1 An Example DEM Mapping

Imagine that two domain independent services need to be executed in sequence. The first service, a 2-D graph layout service, is a member of the 2-D graph domain and the second service, a picture output service, is a member of the output domain. A process executes first the graph layout service and then the picture output service. The data requirements of these two services are very different, which means that some conversion needs to take place between them in order for the first service to use the second service to display the graph on the screen. For a non-trivial example like this, the conversion will probably require user intervention. If the data pre-conditions for the services are both in the form of DEMs as shown in Figure 15 and Figure 17, then nodes could be represented as circles or squares or ovals, and edges as lines. These choices are user choices and will therefore require user interaction.

Simpler conversions may not require user interaction. For example, the conversion from a representation produced by the input domain to a representation used by a sort domain. In this case, the input domain may produce a DIM which is an instance of the DEM shown in Figure 21 and the sort domain may require the DEM shown in Figure 2, which simply requires the provision of another relevant service to convert a sequence of GIF pictures to PCX pictures.

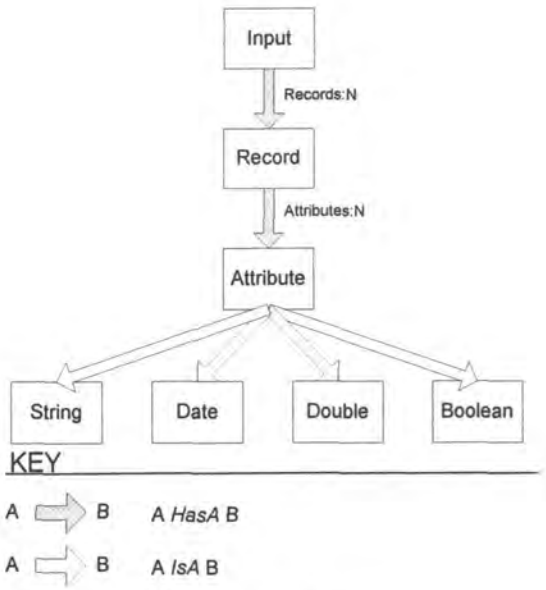


Figure 21 - DEM_{Input}

3.9.2 Determination of Missing Data Entities

Some DEM mappings will not map all source data entities individually because:

- The mapping is a lossy mapping;
- A sub-DEM is mapped whole. This means that the data entities which comprise the sub-DEM are mapped as one entity. Hence, the individual data entities in the sub-DEM do not have individual mappings that describe how they map into a data entity in the target DEM.

Hence, in general, it can't be assumed that the lack of existence of a mapping from a source DEM data entity to a target DEM data entity means that a data entity in the target DEM is missing. However, in some cases, DEM mappings may require a particular source data entity to be mapped and an appropriate target data entity doesn't exist. In this case, there is no DEM entity in the target DEM which is "similar" enough to the source DEM entity in order to allow the source entity to evolve. The mapping process provides an adaptation space for the source DEM, a context in which the evolution can occur. In this case, however, the adaptation space doesn't allow the source DEM entity to evolve because there is no target for the evolution. This may at first seem like a disadvantage. However, the fact that there is no valid mapping tells us two things:

- First, there is a missing data entity in the target DEM which is what the source data entity should evolve into;
- Second, certain characteristics of the target data entity are known, as provided by the chosen data semantics.

3.9.3 Discussion

The ability to encapsulate DEM mappings within a separate, self-contained software entity allows reuse of DEM mappings, in addition to the ability to analyse how they are affected by evolution in other software entities, and how their evolution affects other software entities. Plus, the increased modularity that the extraction of data mapping information

affords software promotes better understanding of software because such information isn't intermingled with other aspects of the code.

Another advantage of expressing data mappings as separate software entities with well-defined boundaries and dependencies is that a service receiving unfamiliar data (in the form of a DIM), or data which isn't an instance of the formal parameters which it's expressed in terms of, will be able to better determine if it can adapt itself to be able to work with the new data. This depends on whether a DEM mapping exists between the newly-arrived data and the data requirements of the service, and whether the mapping is satisfactory with respect to the requirements.

Problems can occur, however, when the mapping from DEM_{input} to the $DEM_{actual\text{-}input}$ breaks down i.e. there is no easy mapping because:

1. The "semantic distance" between the two DEMs is large;
2. The mapping between the two DEMs is complex, is neither obvious nor intuitive.

Additionally, some mappings can be semantic and there will be a number of valid mappings between the two DEMs, but only one will satisfy the user's requirements.

A major difficulty is that of matching the data mapping requirements of a service to the appropriate DEM mapping software entity. This is ultimately achieved through the use of data entity semantics, which are based on:

- *HasA* parent of the data entity;
- *HasA* children of the data entity;
- *IsA* parent of the children (see section 3.9.4.1);
- Services that use the data entity, based on parameter position, use of the data in the service body etc.

Hence, the semantic information of both DEM_{input} and $DEM_{actual\text{-}input}$ are used to determine the appropriate DEM mapping software entity to use. A potential problem stems from semantic heterogeneity: two data entities have different semantic information but are the same data entity; in other words, the same semantic information is represented in different, heterogeneous ways.

3.9.4 Expressing DEM Semantics

There are two approaches to modelling the semantics of data entities in order to aid automated data mapping (discussed in section 3.9.5), both of which are based on the observation that no data entity (or software entity, for that matter) is an island. One can analyse the cohesion of data entities in order to determine how self-contained they are. The use of abstraction to help overcome complexity and the fact that no data entity is completely cohesive means that every data entity will inevitably be related to some set of data entities, through two types of relationship:

- *HasA;*
- *IsA.*

The two approaches to modelling data semantics are:

1. The use of an inheritance hierarchy which describes the semantics of the data entities in terms of where they reside in a super/sub-class hierarchy, much like an object-oriented model. This hierarchy can then be used to determine semantic similarities between data entities in different DEMs that have to be mapped (see Figure 23). For example, it is obvious to a human observer that "record" in DEM_{RDB} and DEM_{Sort} are the same type of data entity. However, this information is not built into a DEM because it is orthogonal to the information modelled by a DEM. Another problem is the typical flatness of inheritance hierarchies and the resultant lack of semantic information that can then be gleaned from the hierarchy. This lack can be made up for to some degree by comparing the children, parent and services that use the data entities in question. A similar approach is used by Kishimoto et al in order to adapt messages when an object's environment (consisting of the other objects with which it communicates) changes [Kishimoto95a];
2. The attachment of attributes to data entities that describe particular aspects of the data entity. The software can reason about the similarity between two data entities in different DEMs by comparing their attributes. For example, the "Edge" data entity in $DEM_{2D\text{-}Graph}$ and the "Line" data entity in the DEM_{Output} may have the attributes shown in Figure 22, which can be compared to determine their similarity. A major problem with this approach is heterogeneity of attributes i.e. different models using different names for the same attribute. This problem can be alleviated by making an assumption: data entities will generally be related only to other data entities in the same domain, which consists of a common ontology of attributes. Hence, by identifying these common attributes and allowing data entities to choose a subset to instantiate with particular values relevant to itself, similarities between data entities based on these attributes can be more easily determined.

Whichever way the semantic modelling is performed, there are problems.

- The success of the automated mapping of data entities is dependent on how the software engineer expresses the semantics of the data entities, and how he relates data entities to each other using either (direct) inheritance semantics and/or (indirect) attribute semantics;
- It may be difficult to relate new data entities to existing data entities. This will be exacerbated by the number of existing data entities, since new data entities will need to be compared with all existing data entities to determine whether or not they are related. Imagine that a software engineer wishes to add DEM_{Record} (shown in Figure 21) to a software system. The process of registering a new DEM involves specifying how each data entity relates to a class hierarchy already known to the system. This allows the software engineer to specify a data entity's semantics by relating its semantics to other data entities in the system. Ideally, the software engineer would want the semantics of "Record" in DEM_{RDB} to coincide with the semantics of "Record" in DEM_{Sort}. This will then result in the automatic mapping algorithm mapping between the two data entities whenever their DEM parents are mapped. However, this

is very contrived and impractical since it requires the software engineer to recognise that the two data entities have the same semantics and to model their semantics accordingly.

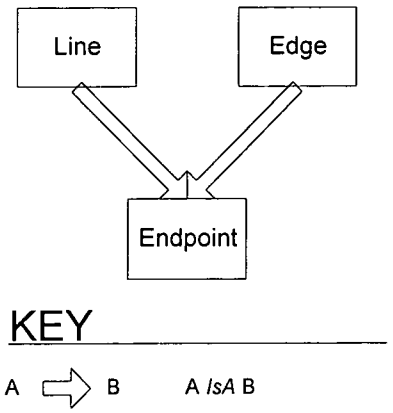


Figure 22 – Example Data Entity Attributes

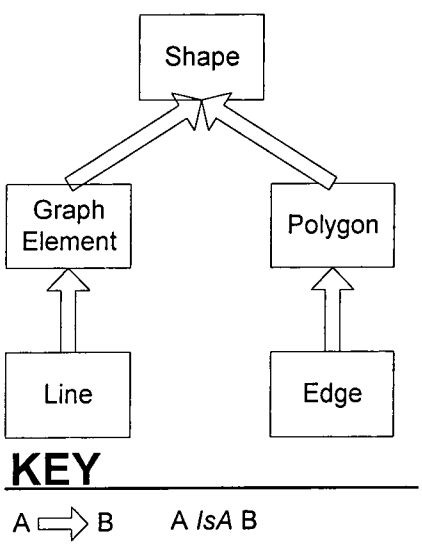


Figure 23 - Inheritance Hierarchy Integrating DEMs Graph and Graphics

Class-hierarchy data semantics is discussed further in 3.9.4.1, and attribute-based semantics in section 3.9.4.2.

3.9.4.1 Class Hierarchy-Based Data Entity Semantics

As shown in Figure 24, class hierarchies form the basis of this method of providing semantics to DEMs in order to allow automated mapping between DEM entities. DEM entities are instances of classes. Common super-classes for two distinct DEM entities indicate that there is some commonality between the DEM entities. This observation can be used to provide a mapping between DEM entities when the mapping is either non-trivial or the user can't or doesn't want to provide a mapping manually. The target DEM provides a reference or context for the mapping, which essentially constrains the mapping process by limiting the target entities onto which the source entities can be mapped. For example, when mapping a 2D graph DEM to a drawing DEM, the 2D graph DEM entities can only be mapped onto the drawing DEM entities – this is a constraint of the mapping.

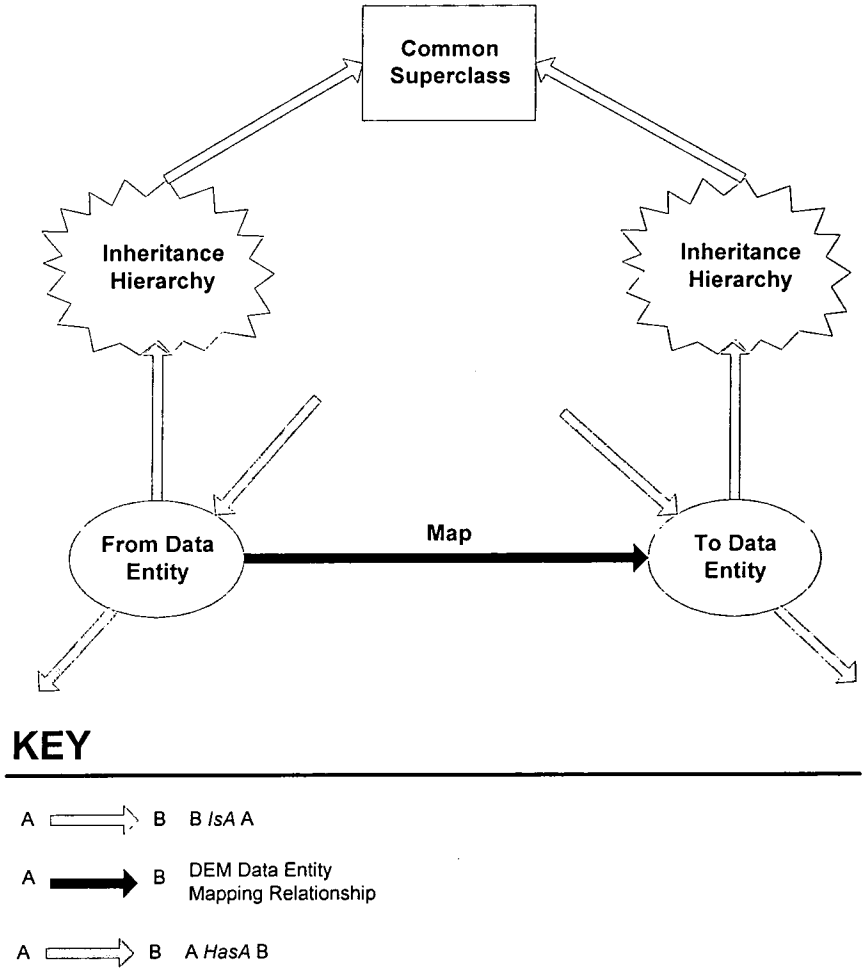


Figure 24 - Class-Based Automated Mapping Between DEMs

A major problem with class-based DEM semantics is that class hierarchies can tend to be quite shallow and therefore provide limited information on similarities between DEM data entities. For example, if the "edge" data entity in $DEM_{2Dgraph}$ and the "line" data entity in $DEM_{Drawing}$ are modelled as shown in Figure 25, then there is no basis for determining that an edge possesses similar characteristics as a line, and therefore that an edge can, with an appropriate (possibly information-losing) mapping, be represented as a line.

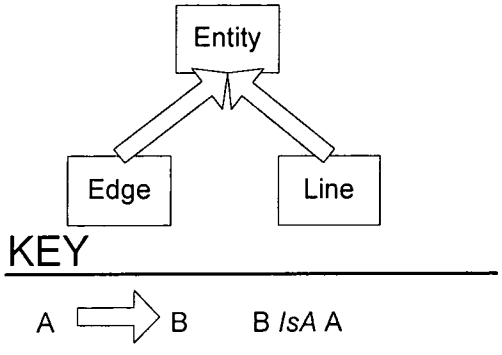


Figure 25 - Shared Characteristics of Edge and Line

This mapping process can also be viewed as a form of evolution in which the source DEM evolves into the target DEM. The main difference is that the form of the evolved data entity is known. What isn't known is how software entities in

the source map to software entities in the target. This is different from functional evolution in the sense that the target of evolution is known, but only at a high level of abstraction. The difficulty lies in actually mapping this abstract requirement into the software entities of the software.

The mapping context described in the previous paragraph has another advantage. Consider mapping an arbitrary source DEM data entity. The main characteristic of the target DEM data entity is that it must belong to a particular DEM. This implies that the target DEM entity must have certain characteristics (inherited from the target DEM). For example, in the 2Dgraph-Drawing mapping, for an arbitrary 2Dgraph entity, the target entity must be part of a drawing. This means that one of its characteristics is that it is a shape. Information such as this and characteristics of the source DEM entity can be used to determine the characteristics of the required target DEM entity. For example, in mapping a 2Dgraph.Edge entity into $DEM_{Drawing}$, information about the Edge entity such as:

- Has a from co-ordinate (inherited from one of its 2DGraph.Node children);
- Has a to co-ordinate (inherited from its other 2DGraph.Node child).

can be used to ascertain that the target entity must have similar characteristics. The target must also be a shape, as determined from the fact that the target DEM is $DEM_{Drawing}$, which consists of shapes. This leads on to a potential advantage: if the mapping fails, the required data entity can at least be partially characterised in order to aid the software evolution process.

Class-hierarchy data semantics will require multiple inheritance. For example, imagine mapping between $DEM_{2D\text{-}Graph}$ and DEM_{Sort}. Unlike mapping between DEM_{RDB} and DEM_{Sort} where the semantics of the "Record" data entity in both DEMs are the same, there is no "Record" data entity in $DEM_{2D\text{-}Graph}$. Therefore, which data entity (or data entities) should be mapped? This is dependent on user requirements and so will be determined by how the software engineer models the data entities in both DEMs. However, it is clear that data entities will need to have more than more parent class, because an individual data entity may be mapped to many other different data entities, themselves each having very different semantics. Imagine that the "Edge" data entity in $DEM_{2D\text{-}Graph}$ is to be mapped to the "Record" data entity in DEM_{Sort}. This would require that both data entities share a common parent class. If the "Edge" data entity were then to be mapped to another data entity, they too would need to share a common parent class.

3.9.4.2 Attribute-Based Data Entity Semantics

Attribute-based data semantics are very similar to semantic networks; both approaches utilise the notion of entities which are related by an unconstrained number of relationships. In the case of attribute-based data semantics, the attributes are not data entities themselves, but attribute entities which aim to describe indirectly how data entities are related by virtue of a number of data entities possessing the same attributes.

As discussed previously, the difficulty with attribute-based data semantics is the potential for heterogeneity creeping into the model. Another problem is that of "semantic distance", which means that data entities (the concept can be extended to that of software entities in general, with appropriate substitution of semantics; for example, service semantics are

different than data semantics) are related, but their relationship is complicated. One approach to dealing with this is to utilise a common pool of attributes which describe the domain which encompasses all existing and future data entities. This, at first, seems constraining. However, this assumption can be justified by considering the fact that, if data entities are constrained to a particular domain and domain attributes tend to be fairly static, then new data entities can be described in terms of this pool of domain attributes.

3.9.5 Automating DEM Mappings

The lack of any explicit modelling of data semantics in traditional software makes automated mapping difficult. Even classes, with enhanced semantic modelling capabilities than some other data models, don't provide much in the way of semantics for what they are modelling. In addition, successful data semantics depends on:

- Getting the terminology right and ensuring that semantics isn't represented differently in different parts of the software system;
- Richness of semantics. The semantic model is rich enough to ensure that automated mapping is successful. This is difficult to achieve because real-world concepts can be complex, type relationships difficult to determine and dynamic in nature. Take, for example, the graph node data entity. Semantically, this is part of the graph domain but can be represented in other domains too, such as the drawing domain in which it can be represented as a square or a circle. Hence, semantically and in order to aid automated mapping, graph nodes are potentially circles and squares. Failure to model this information in the data model results in an inability to automatically map graph node data entities to circle or square data entities.

Comparing two classes for similarity could be performed by comparing both the structure of the data they encapsulate and the methods they encapsulated. However, how can methods be compared for similarity? Comparison based on parameters is not enough because it doesn't take into account the behaviour of the method. Perhaps the comparison could be based on the actual code components, so that a similarity test results in a fuzzy logic result based on use of similar procedure calls, but no theory exists for this.

DEM mappings provide a way of encapsulating how a DEM evolves into another DEM. However, DEM mappings typically require user input to specify the mappings, because the representation of data semantics is typically very limited. Can the automation of data mappings be improved by modelling the semantics of the data entities in DEMs? This is not the case with traditional software. Inheritance in OO software provides some limited semantics, since classes can be compared based on the inheritance hierarchy, but the quality of the semantics is inevitably dependent on the modelling of the software engineer and it isn't clear what the best semantics are.

The automatic mapping algorithm (as shown in Figure 26) requires that every data entity in the source DEM be checked against every data entity in the target DEM, in order to check for dependencies that could result in mapping occurring. This is a potentially time-consuming process that increases in time complexity quite drastically as the number of data entities in either DEM increases. This problem can, however, be aided by parallelising the algorithm and running on multi-processor hardware.

For every data entity pair (SDE, TDE), such that SDE is a member of the set of source data entities (which may constitute the whole source DEM or just a part of the source DEM, depending on the mapping requirements), and TDE is a member of the set of target DEM data entities:

Let $SDE_{Children}$ = {Primitive children of SDE, through *HasA*, *IsA*, and *UsedBy* <*Service*> relationships}
Let $TDE_{Children}$ = {Primitive children of TDE, through *HasA*, *IsA* and *UsedBy* <*Service*> relationships}

Try and match SDE*Children* and TDE*Children*.

Figure 26 - DEM Mapping Rule

It is interesting to speculate on the possibilities for negotiation of data mappings between services, provided appropriate data semantic machinery is in place as discussed briefly above. A particular problem occurs when there are many potential mappings. The chosen mapping will typically be based on user requirements, which can't be determined purely from data semantics. In this case, the negotiation protocol adopted may have to include interaction with the user. The success of the negotiation protocol would necessarily be constrained by the quality of the data semantics model employed.

4 Data Evolution Spaces

4.1 DEM Evolution

DEM evolution is concerned with how DEMs and DIMs can be changed, the types of change that can occur to them. The aim of this section is to produce a taxonomy of changes that provides an ontology for talking about changes to DEMs and DIMs, which in turn allows one to map change types to their effects on other software entities.

It is also important to point out that both adaptation evolution and integration evolution can be applied to data. Adaptation evolution is performed in a DEM by means of refinement of *existing* data entities and relationships. Integration evolution is performed by adding new data entities and relationships.

4.1.1 DEM Generalisation and Specialisation

Changes to a DEM can be viewed in terms of two types:

- The target DEM (after evolution) is a generalisation of the source DEM;
- The target DEM is a specialisation of the source DEM.

A DEM, DEM_B, is a generalisation of a DEM, DEM_A, if DEM_B can be used wherever DEM_A is used without causing any ripple effects. In other words, DEM_B doesn't introduce any assumptions which invalidate its use by any other software entities (for example, services and DEMs) that use DEM_A. There are two types of generalisation:

- **Global generalisation**: this type of generalisation doesn't affect any existing or new software entities. This can be achieved in two ways:
  - The introduction of an *IsA* relationship;
  - Re-configuration of the DEM such that the data entities stay the same but the *HasA* relationships change within the full connectivity of the DEM. A fully-connected DEM is a DEM in which all potential *HasA* relationships are modelled in the DEM (even if this introduces redundancy into the DEM), as shown for $DEM_{2Dgraph}$ in Figure 27, and is DEM-dependent (i.e. it doesn't necessarily mean that every data entity is related to every other data entity, but is a semantic consideration and therefore domain-dependant). Hence, a fully-connected DEM is a global generalisation, as is any DEM which is a valid subset of a fully-connected DEM. A valid subset of a fully-connected DEM is again DEM-dependent. In this case, valid subsets include at least (1), (3), (4) or (2), (5);
- **Partial generalisation**: this type of generalisation is guaranteed not to affect any existing software entities, but may affect new software entities. In other words, the generalised DEM doesn't break any assumptions for existing software entities but makes no guarantees about new software entities.

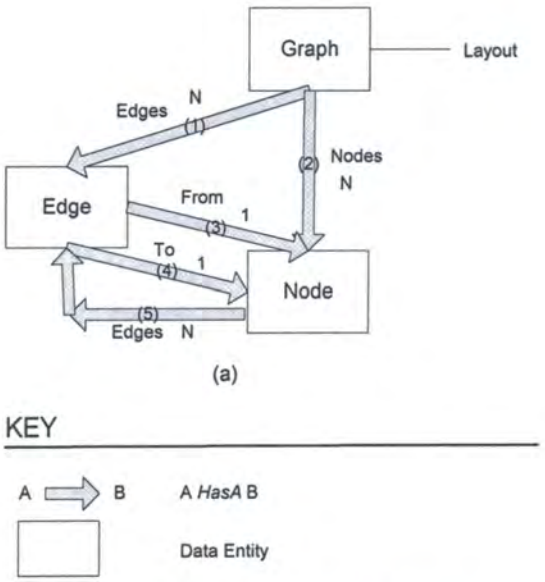


Figure 27 - A Fully-Connected $DEM_{2DGraph}$

4.1.2 A DEM Change Type Taxonomy

Hursch identifies a set of evolution operators and their targets within the context of object-oriented models [Hursch95a]. The operators are:

- Add;
- Remove;
- Rename.

The targets of these operators are:

- Classes;
- Reference relationships;
- Inheritance relationships.

Their cross-product provides a basic set of adaptation types, which can be composed to provide higher level adaptation types. For example, the movement of a reference from a super-class to each sub-class is equivalent to the composition of the removal of the reference from the superclass, and the addition of a reference to each sub-class. This type of classification provides a taxonomy of changes to a model that is sufficient to describe all possible changes to that model. In the case of object-oriented models, there is a basic set of changes upon which higher-level changes can be built. The existence of such a taxonomy provides a way to analyse the adaptability and ripple effects of software because change types can be mapped to the effects of that change on the software. Hence, some change types may not produce ripple effects because of the inherent adaptability of the software with respect to that change. Other change types may produce ripple effects.

The evolution operators:

- Add;
- Remove, and;
- Change.

can be applied to DEMs in order to provide a set of adaptation types for DEMs. Having just add and remove operators on their own is not enough for the following reason. Consider the movement of a reference from a class to each sub-class, a change type that Hursch calls distribution of common reference [Hursch95a]. Services should be adaptable to this type of change if carried out as an atomic operation because the information modelling power of the data model doesn't change i.e. the changed model is essentially equivalent to the original model before the change. However, expressing this change as two operations (remove reference followed by add references in each subclass) fails to capture the fact that the target of the relationship is essentially the same reference. Furthermore, removal of a data entity would immediately cause problems because services are not adaptable to removal of data from a data model. Hence, the "change" operator captures the essence of a change type which involves more than simply a combination of addition and removal operations.

The targets for these evolution operators are:

- Whole DEMs;
- Data entities;
- Data entity relationships:
  - *HasA*;
  - *IsA*.

The cross-product of the evolution space operators and their targets produces the set of data adaptation types shown in Table 4.

Data Adaptation Type	Reference (Section Number)
Add a New DEM	4.1.5
Remove an Existing DEM	4.1.8
Add a New Data Entity	4.1.6
Remove an Existing Data Entity	4.1.7
Add a New *HasA* Relationship	4.1.3
Remove an Existing *HasA* Relationship	4.1.4
Add a New *IsA* Relationship	4.1.9
Remove an Existing *IsA* Relationship	4.1.10

Table 4 – DEM Adaptation Types

4.1.3 Addition of a New *HasA* Relationship

The addition of a new HasA relationship to a DEM potentially creates more DEM paths. No existing DEM paths are affected by this operation. In addition, no services dependent on the DEM are affected because the operation produces a generalisation of the original DEM.

4.1.4 Removal of an Existing *HasA* Relationship

The removal of an existing *HasA* relationship will affect any DEM paths which are dependent on it (see section 5.4). It will also lead to the removal of any data entities for which the relationship is the last *HasA* parent.

4.1.5 Addition of a New DEM

Since the evolution of data occurs as a result of changes in requirements and requirements can conflict, data evolution can potentially cause conflicts. An example of this is data evolution that involves the addition of a new DEM, which may encapsulate constraints that conflict with existing DEMs in a software system. Hence, the creation of new DEMs may have an effect on or may be affected by existing DEMs in the software. For example, an important DEM in a simple telephone switch is the "ActiveCalls" DEM, which identifies which users are currently in active calls with each other i.e. which users are currently connected to one another. The creation of an "OnHold" DEM will cause problems because it shares the same primitive data type (telephone number) with the existing "ActiveCalls" DEM. In effect, there are shared constraints between the two DEMs which are inherently part of the modelling process. However, there is an heuristic which allows one to determine if conflicts may exist between an existing DEM and a new DEM: if a data entity is shared between the two DEMs then this may cause conflicts.

4.1.6 Addition of a New Data Entity

Adding a new data entity to a DEM can affect:

- DIMs which are an instance of the DEM;
- DEM paths which use the DEM.

An important aspect of DEM changes is the ability to identify the characteristics of changes which may affect dependent software entities. One such characteristic is concerned with how changes to the DEM affect the domain stability. For example, the characteristics of a change to a DEM may fall into one of the following two categories:

- The data evolution changes the domain, which inevitably affects dependent software entities such as services and DEM mappings. For example, adding a new co-ordinate (or dimension) in the 2-D graph domain invalidates existing graph layout algorithms and changes the domain into the 3-D graph domain;
- The data evolution doesn't change the domain and has little effect on dependent software entities. For example, adding a new field to a record in the database domain.

The former types of change are likely to lead to new functionality being added to the DEM, whilst the latter are not. For example, adding a new co-ordinate to the 2-D graph domain leads to a 3-D graph domain and existing services, which are specific to that domain such as graph layout algorithms, will need re-writing. This is unavoidable since services encapsulate domain expertise which needs to be encoded in some way by a software engineer. Software cannot introduce new behaviour automatically, unless that behaviour already exists in some form and, in addition, there is some way for the software to relate the new behaviour with both the existing behaviour and new requirements through, perhaps, some form of reflective model.

4.1.7 Removal of an Existing Data Entity

Removal of a data entity affects all relationships to which it is connected. These relationships include:

1. *IsA* relationships;
2. *HasA* relationships;
3. *InstanceOf* relationships;

The effects of data entity removal on *InstanceOf* relationships is discussed in section 5.2.1 on DIM adaptability. As for *IsA* and *HasA* relationships, an approach similar to Hurschs' for object-oriented models [Hursch95a] can be adopted, provided that the data entity is a child in an *IsA* relationship. Hursch overcomes class removal through what he calls "telescoping of inheritance" [Hursch95a], which moves all references and elements in the class to be removed into *all* subclasses of this class. In the case of DEMs, the removed data entity's sub-model can be moved to the *IsA* parent, as shown in Figure 28. In effect, those *HasA* and *IsA* relationships and data entities which are related to the removed data entity are moved into any *IsA* parents of the data entity.

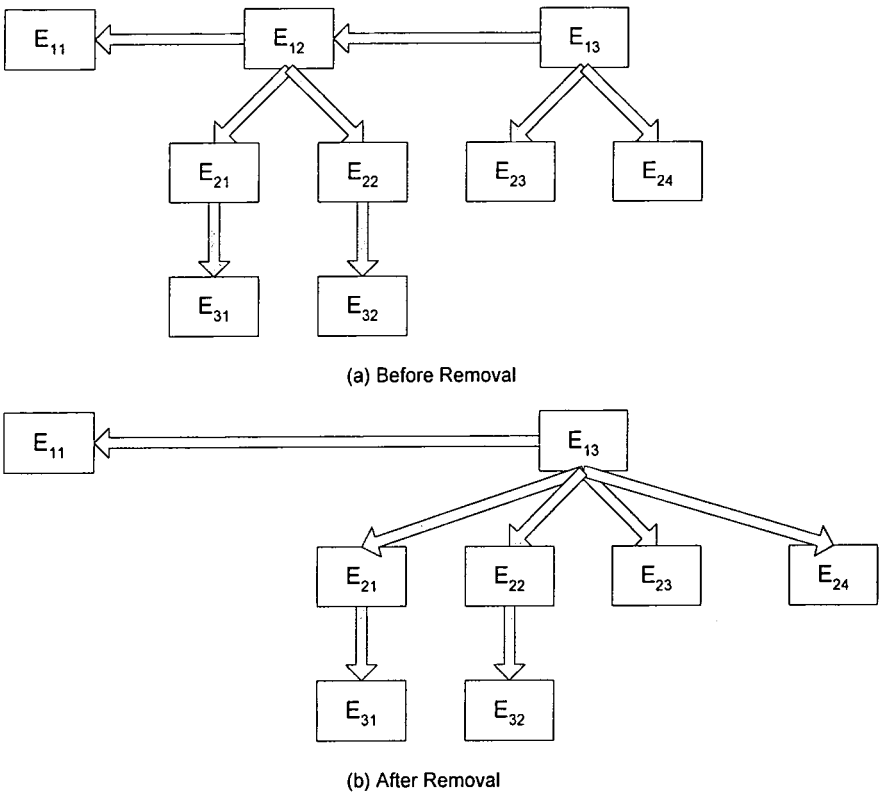


Figure 28 - Data Entity Removal

The removal of a data entity is a capacity-reducing transformation and, as such, has the potential for affecting any dependent software entities such as services. This is discussed in the relevant sections on software entity evolveability.

4.1.8 Removal of an Existing DEM

The removal of a whole DEM involves removing all data entities and *HasA* relationships from the DEM, as well as any *IsA* relationships in which data entities in the DEM are involved. This is equivalent, in an object-oriented model, of removing a whole class and all of its members. The removal of a DEM can affect any of the relationships (and therefore software entities) to which the DEM is connected, such as:

- *Uses/Removes/Updates/Produces*;
- *InstanceOf*;
- *IsA*;

The effects of DEM removal on the first two types of relationship is discussed in section 3.1.1 (chapter 7) and section 5.2.1, respectively. With respect to the *IsA* relationship, a decision has to be made about whether or not any *IsA* parents of the removed DEM should "inherit" the data entities in the DEM. This will be the case if any of these parents extend the removed DEM in some way, in which case they will be dependent on the structure of the removed DEM. Failure to move removed data entities to *IsA* parents may render these parents inconsistent and have effects on any services which utilise these removed data entities.

4.1.9 Addition of a New *IsA* Relationship

The addition of a new *IsA* relationship produces a generalisation of the original DEM, providing that the change doesn't conflict with any existing *IsA* relationships. If this is the case, no existing services should be affected

4.1.10 Removal of an Existing *IsA* Relationship

Removal of an existing IsA relationship is a lossy operation (see section 4.1.11) and results in loss of information. The effects of addition and removal of *IsA* and *HasA* relationships on the interface of a DEM is discussed further in section 4.1.11, which provides a mapping between changes in the DEM and the effects on its interface.

4.1.11 Characteristics of DEM Change Types

An important characteristic of DEM change types is the relationship between the changed DEM and the initial DEM with respect to modelling power. Some changes may only change the structure of the DEM without changing the modelling power of the DEM, as Hursch has recognised for object-oriented software changes [Hursch95a]. Table 5 shows the three types of relationship that exist between a DEM before and after evolution with respect to this notion of modelling power. In addition, the table shows the types of change which fall into each category.

Type	Description	Change Types	Example
Extension	The target DEM is a generalisation of the source DEM.	Add a new data entity. Add a new *IsA* relationship. Add a new *HasA* relationship.	Mapping from a singly-linked list to a doubly-linked list.
Equivalence Class	The target DEM is equivalent in modelling power to the source DEM.	Add and remove *HasA* operations. Typically domain-dependant. *No* removal, addition or adaptation of existing data entities.	The mapping shown in Figure 29.
Lossy	The target DEM is a specialisation of the source DEM. This implies a loss of modelling power and a consequent effect on clients.	Remove an existing data entity. Remove an existing *IsA* relationship. Remove an existing *HasA* relationship.	Mapping from a doubly-linked list to a singly-linked list. See Figure 30.

Table 5 - Data Mapping Types

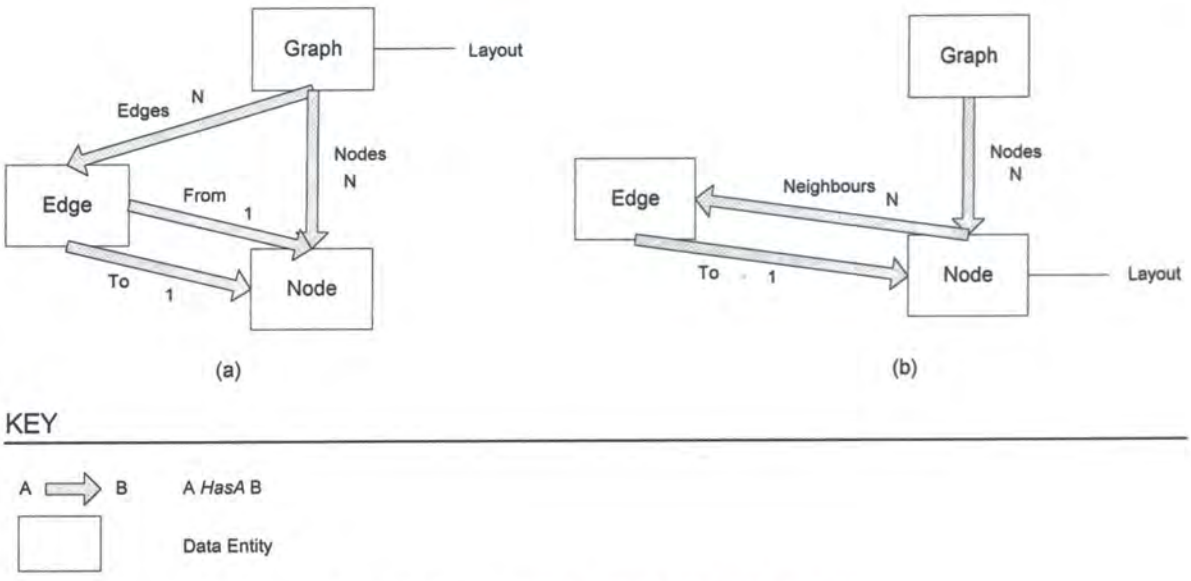


Figure 29 – DEM Evolution Caused by Service Evolution

As an example of a lossy data mapping, consider the data mapping shown in Figure 30, where the removal of the "Previous" node attribute results in any DIMs that instantiate DEM_{DLL} losing data.

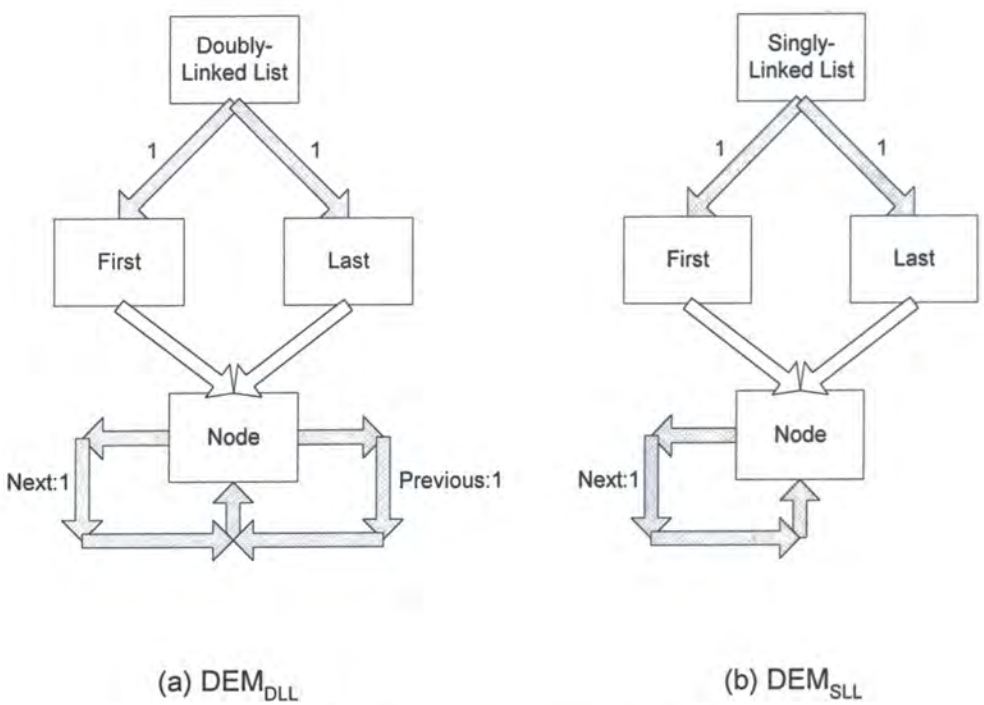


Figure 30 - A Lossy Data Mapping

4.2 DIM Evolution

Although instance evolution has been covered generically in chapter 5 section 3.5, DIM evolution is a special case because data instances may change independently of the DEM of which they are an instance. Table 6 shows the types of DIM evolution which can occur.

DIM Change Type	Comments
Add a Data Instance	See section 4.2.1.
Remove a Data Instance	See section 4.2.2.
Adapt a Data Instance	See section 4.2.3.
Add a DIM	See section 4.2.4.
Remove a DIM	See section 4.2.5.

Table 6 - DIM Evolution Types

4.2.1 Add a Data Instance

This is an extensional change and corresponds to a change in a variable's value in a software language. It shouldn't result in evolution unless changes occur to the DIM which are outside the "interface" of the DIM (imposed by its DEM), which means that the change has turned into an intensional change because:

- The requirements have changed, resulting in new data modelling requirements and an evolved DEM. For example a client service, C, requires a graph to be laid out using a 2D graph layout server service, G. The graph DIM to be laid out, however, contains 3D graph data. In essence, C's requirements of G have changed, resulting in new data instances in the DIM and, hence, ripple effects on G;
- The modelling of the DEM has failed to capture the required semantics of the original requirements fully enough. In this case, the contract between two services using a particular DEM, for example, hasn't fully captured the data modelling requirements of the client service. The server service implements the requirements of the client based on this incomplete contract. Another problem is the inability of DEMs to model semantics completely, which results in client and server services having different views or semantics of the DEM;
- The client service, C, and server service, G, both assume capabilities of the DEM which it doesn't possess because it doesn't satisfy its requirements. An extreme example is:
    - The requirements are for a DEM which allows 2D graph data to be modelled along with a weighting for each node. This information is to be used in any graph layout services to alter the behaviour of the service.
    - The DEM models 2D graph data but fails to allow the modelling of node weightings.
    - The client requests the server to lay out a DIM, which is an instance of this DEM.

  This will causes problems if the client service passes weighting information because this information will be lost when the graph layout service is called.

4.2.2 Remove an Existing Data Instance

Surprisingly, this kind of change may cause ripple effects. Consider again the example of the graph layout server described in the previous section, and imagine that in this case the Y co-ordinate data instances are removed from the

DIM. This results in G being unable to interpret the DIM, because it doesn't satisfy the *InstanceOf* relationship with respect to its DIM any longer. Of course, these kinds of changes can be forbidden in order to ensure consistency of the *InstanceOf* relationship and, by implication, any assumptions which services make about the DIM (services assume any DIMs they use will be instances of the relevant DEM which they are implemented in terms of. If the *InstanceOf* relationship is consistent, then changes in DIMs shouldn't cause any ripple effects in the service).

4.2.3 Adapt a Data Instance

Adapting a data instance means changing its instance submodel whilst ensuring consistency with respect to the DEM of which it is an instance. If this last condition is met, no ripple effects should occur. Ripple effects will occur, however, if the *InstanceOf* relationship is broken by the adaptation.

The real problems occur, however, when changes in a DIM are consistent with respect to the *InstanceOf* relationship but still cause ripple effects because of assumptions which services make about the DEM. This is especially true of existing software languages. For example, a C type of the form "char *" is very abstract and can be used to represent many things, such as:

- A block of memory;
- A part of a linked list;
- A part of a more complex pointer-based data structure.

This makes it difficult to determine the consistency relationship between the data instances and the data entity. DEMs improve upon this state of affairs by constraining the data model and making explicit the structure of the data. In effect, strong typing is used to ensure that consistency can be checked.

4.2.4 Add a DIM

Adding a new DIM shouldn't in itself cause ripple effects unless it is then to be used by, for example, a service instance which then requires new capabilities in order to interpret it.

4.2.5 Remove a DIM

The reflective model provides trace-ability of the production, removal, and usage of DIMs and data instances in a software system through the *Produces*, *Removes*, *Uses* and *Updates* relationships. Hence, the effects of DIM removal on other software entities in a software system can be traced through the *Uses*, *Removes* and *Updates* relationships, and a new *Produces* relationship created in order to lessen the impact of the removal.

4.3 DEM Mapping Evolution

Table 7 shows the types of DEM mapping evolution which can occur.

Evolution Operator	Mapping
Add	Adds a new data entity to a data entity mapping.
Remove	Removes an existing data entity from a data entity mapping.
Adapt	Changes the semantics of an existing data entity in a data entity mapping i.e. it maps it to something else.

Table 7 - DEM Mapping Evolution

5 Data Evolveability

5.1 DEM Adaptability

Data adaptability is concerned with the ease with which data structures (DEMs) can be changed to meet new representation requirements, and how well DEMs adapt to changes in software entities on which they depend, which basically includes other DEMs. This section is concerned with forms of DEM adaptability.

5.1.1 Adaptability With Respect to DEM Evolution

A DEM is related to other DEMs through *HasA* and *IsA* relationships. Changes in a data entity may have ripple effects on any data entity related to the data entity through IsA relationships, due to a dependence of the parent on the child's structure. This is a known problem in the field of object-oriented research, termed the "Fragile Base Class Problem", in which inheritance creates a dependency of a subclass on the internal structure of its superclass(es). Changes in a class may affect its subclasses. The "Fragile Base Class Problem" doesn't affect data entities related to the changing data entity through *HasA* relationships because of the inherent semantics of the *HasA* relationship, although services using this data entity may indeed be affected.

Mikhajlov et al approach this problem through the use of what they term "disciplined inheritance", which is based on placing particular constraints on the inheritance relationship [Mikhajlov97a]. Most problems stem from the inherent contract created between a class and its dependants, which are of a number of types:

- Dependence on internal class structure;
- Creation of undesirable side-effects. For example, mutual recursion;
- Dependence on behaviour;
- Dependence on undocumented/unknown/implicit assumptions.

As an example, consider the class model shown in Figure 31 in which a subclass (or client), E, of class, C, makes a set of assumptions about C (such as the set of dependencies between the internal elements of C, which are depicted by the *Uses* and *DependsOn* relationships in the figure). When C evolves into C', these assumptions may be invalidated. In this case, the evolution of C into C' and the resultant *IsA* relationship between E and C' produces a dependency between X and Y which conflicts with a no-dependency assumption made by a method in E.

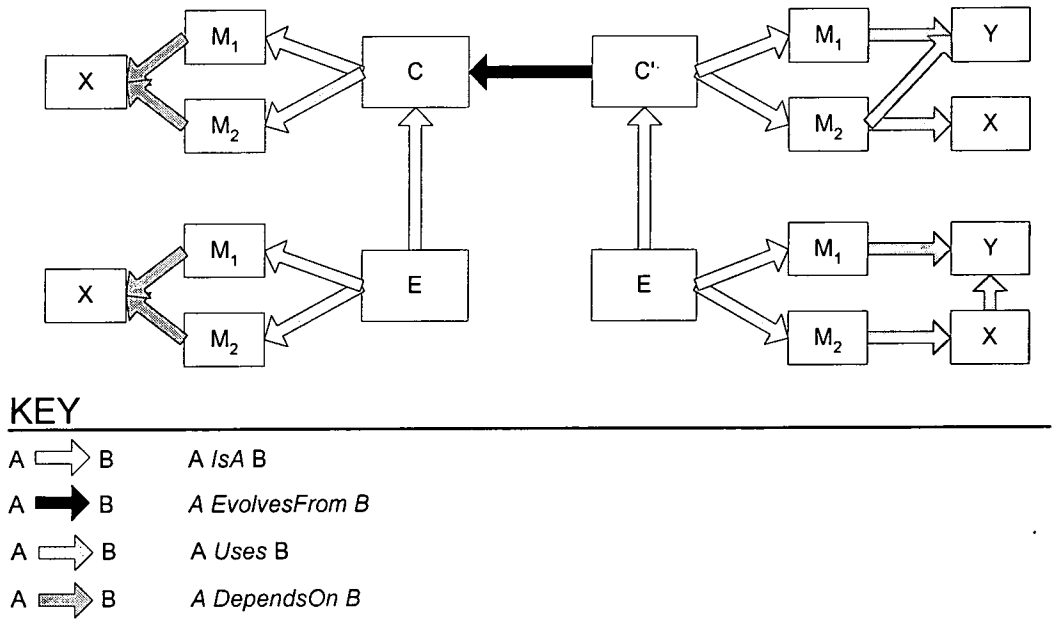


Figure 31 – The Fragile Base Class Problem

In general, changes introduced in C' which break the interface (consisting of the explicit interface and implicit assumptions) that C supplies to its clients, will result in ripple effects in these clients which must then adapt to the new interface. These changes typically involve changes to *existing* aspects of C, although changes which extend existing aspects can also produce ripple effects. For example, extensions of the behaviour of a method which break the requirements of clients. Examples of changes to existing elements of a DEM that can produce such ripple effects are:

- Changes that specialise the data in some way;
- Changes to method behaviour that:
  - Break assumptions;
  - Remove behaviour;
  - Introduce new behaviour which breaks the requirements of clients of the method.

In contrast, changes of the following types should not produce ripple effects:

- Changes that generalise existing data;
- Changes to methods/behaviour which don't break requirements. If the requirements are very strict, the range of changes to the behaviour which don't produce ripple effects will be small, in comparison to looser requirements which allow for a greater range of changes to the behaviour. For example, real-time applications place strict requirements on the speed of a function. A word-processing application, in comparison, places less strict requirements. A major problem with existing programming languages, models and architectures is that the mapping between code entities and the requirements they implement is difficult to determine, so that tracing the effects of a change on the requirements is difficult to do.

5.2 DIM Adaptability

This section analyses DIM adaptability and, in particular, the effects that changes in the DEM of which a DIM is an instance have on the DIM.

5.2.1 Adaptability With Respect to DEM Evolution

Much of the research into the dependence of data instance on data entity models, and of the characteristics of their relationship, has been carried out within the database community [Clamen94a]. There has been little work on data instance model adaptability with respect to data entity model evolution, because most of the work has focussed on what happens to the data instance model in response to types of change in the data entity model, such as data entity removal, addition and change. In current software languages, type checking provides a mechanism to check for the consistency of data instance models (variables) and data entity models (types). Using type checking in existing software languages, the types of ripple effects caused by changes in a type on variables of the type are shown in Table 8. These ripple effects can be determined at compile time. Note that the addition of a new element to a type will not be detected by the type checker with respect to variables (although software languages will generally detect the effects of type changes on the formal and actual parameters to functions). This is not always desirable, since the addition of new elements to a type may change the semantics of the type and produce ripple effects on variables. For example, a 2D graph type evolves into a 3D graph type and in the process changes the semantics of the type. Any variables of this type will now be valid with respect to any type checking mechanism, but will be invalid semantically. A type checker can't detect this type of ripple effect because they are semantic and rely on the software engineer to be determined. In addition, current software languages fail to keep variables and types consistent so that, in the case of the 2D graph type evolving into a 3D graph type, any variables will not contain an instance for the third co-ordinate.

Type Change	Caught By Type-Checker
Add new data entity	No
Remove existing data entity	Yes
Adapt existing data entity	Yes

Table 8 - Type Checking and Data Model Ripple Effects

The high dependence of a DIM on the DEM of which it is an instance results in high context dependence, or coupling between the two. This occurs because of the inherent dependence of DIMs on DEM structure, as pointed out in section 3.5, which reduces the adaptability of DIMs with respect to DEM evolution.

A potential solution to improved DIM adaptability is to base the dependence of DIMs on another kind of DEM interface rather than their structure. For example, based on an abstraction of the DEM structure, or based on the use of DEM paths. The latter, however, will only improve the adaptability of DIMs with respect to particular types of change. For example, changes in the structure of the DEM could be adapted to because information isn't lost from the DEM (the DEM paths used by the DIM don't depend on DEM structure and so will not be affected by DEM structure changes). This, in turn, means that DIMs will not be affected by DEM structure changes . When information is lost from a DEM,

however, adaptability suffers. This is an unavoidable problem and can only be overcome by careful representation of the dependencies between DIMs and DEMs, or the prevention of removal of data entities from a DEM which is an undesirable option.

DIM adaptability with respect to the following DEM change types has to be considered:

1. Data entity addition;
2. Data entity removal;
3. *HasA* relationship addition;
4. *HasA* relationship removal;
5. *IsA* relationship addition;
6. *IsA* relationship removal.

Change types 5 and 6 don't affect the DIM, because DIMs only depend on data entities and *HasA* relationships. Changes in *IsA* relationships may, however, affect other DEMs.

The addition of a new data entity to a DEM (change type 1 in the list above) can have an effect on any DIMs which are an instance of this evolving DEM. Consider the DEMs:

- Edges = ((Edges *HasA* Edges Edge N) (Edge *HasA* Node1 int 1) (Edge *HasA* Node2 int 1));
- Nodes = ((Nodes *HasA* Nodes Node N) (Node *HasA* X float 1) (Node *HasA* Y float 1)).

where the notation is (<Data Entity> *HasA* <Attribute> <Cardinality>), and the DIMs "ExampleEdges" and "ExampleNodes" of which they are respectively instances. The state of these DIMs at a particular point in the execution of a program is:

- ExampleEdges = ((ExampleEdges $HasA^I$ Edge1) (ExampleEdges $HasA^I$ Edge2) (Edge1 $HasA^I$ 1) (Edge1 $HasA^I$ 2) (Edge2 $HasA^I$ 2) (Edge2 $HasA^I$ 3));
- ExampleNodes = ((ExampleNodes $HasA^I$ Node1) (ExampleNodes $HasA^I$ Node2) (ExampleNodes $HasA^I$ Node3) (Node1 $HasA^I$ 100) (Node1 $HasA^I$ 100) (Node2 $HasA^I$ 200) (Node2 $HasA^I$ 200) (Node3 $HasA^I$ 300) (Node3 $HasA^I$ 300)).

where the notation is (<Data Instance> $HasA^I$ <Data Instance>|<Primitive Data Instance Value>).

Now, consider a new requirement which means that the edges to which a particular node is related can be determined. This is possible with the existing DEM, but would result in inefficient lookup. By trading off increased memory memory usage for lower speed, the DEM can be evolved in order to allow this information to be determined more quickly. This simply requires that the "Node" DEM evolve to include an array of Edges, as shown:

- Nodes = ((Nodes *HasA* Nodes Node N) (Node *HasA* X float 1) (Node *HasA* Y float 1) (Node *HasA* Edge N)).

However, this type of change produces an inconsistency between the DIM and DEM unless the DIM is adapted after evolution of the DEM by adding appropriate data instances to the DIM. In this case, the existing "ExampleNodes" DIM will need new data instances. The structure of these data instances and the values of any new primitive data instances may be determined from the existing DIM data. In this example, the content of each edges array is dependant on the edges to which each node is related, which can be determined from the existing DIM data.

For change type 2, a decision needs to be made about what to do with data instance information when the data entity information on which it depends is lost because the data entity is removed. There are a number of options:

- The data instance information is lost or removed in order to retain the consistency of the data model as a whole;
- The data instance information is retained, but the link to the DEM is severed in order to maintain the consistency of the data model as a whole;
- Data entity removal operations are prevented. This is problematic for situations when removal of the data entity is required. A potential solution to this is to adopt three states for each data entity:

    - Present;
    - Not present;
    - Removed – flagged as removed.

    This allows data entities to be present and still connected to the DEM of which they were once an active member thereby permitting trace-ability between different DIM and DEMs versions, but no longer affect any *new* DIMs. This is similar to work on source code control tools, such as SCCS [Bolinger95a] and RCS [Bolinger95a], which provide a trace-ability mechanism between source code entities.

The addition of a new *HasA* relationship to a DEM will affect any DIMs which are an instance of the evolved DEM. However, the required adaptation of the DIM is a simple one because it is in direct correspondence to the mapping in the DEM.

As discussed before, the adaptability of instances with respect to evolution in the software entity of which they are an instance is difficult to achieve because of the high context dependence (or coupling) of instances on software entities. This is also true of DIMs and DEMs. However, by utilising DEM paths in the DIM model, the adaptability of DIMs with respect to DEM evolution can be improved since DEM paths shield changes in the *structure* of the DEM from dependent software entities, including DIMs. This mapping between data instances and data entities can be encapsulated within the DIM *InstanceOf* DEM relationship in the reflective software entity model.

5.3 DEM Mapping Adaptability

5.3.1 Adaptability With Respect to DEM Evolution

The dependence of DEM mappings on DEMs is very high, much like the dependence of DIMs on DEMs. An analysis of the effects of DEM evolution on DEM mappings begins with types of evolution in the source DEM:

- Add data entity: this type of change shouldn't affect *existing* data entity mappings in the DEM mapping, but may require adding new data entity mappings. This can not be automatically adapted to and requires user intervention;
- Remove data entity: this results in the removal of data entity mappings from the DEM mapping, and the removal of mappings;
- Adapt data entity, which means changes in the structure of the *HasA* and *IsA* relationships involving the data entity. Removal of *HasA* relationships implies removal of dependent parts of the DEM mapping which use the HasA relationship. The addition of *HasA* relationships implies the addition of new data entity mappings to the DEM mapping. The addition and removal of *IsA* relationships can't be adapted to, in general, because of the semantic considerations; adding a new *IsA* relationship may open up new possibilities for mapping, whilst removal of an *IsA* relationship may affect existing mappings.

5.4 DEM Path Adaptability

DEM paths provide a way to encapsulate the structure of a DEM and shield changes in DEM structure from software entities which are dependent on DEM structure. This is a form of evolution localisation, characterised by the fact that the change in structure results in a different implementation or representation which still satisfies the original requirements.

The addition of a new data entity can affect any dependant DEM paths in the following ways:

- **Add a data entity as a non-leaf data entity:** this type of change will affect an existing data entity path and so may affect existing services [Hursch95a], resulting in a need to adapt the services to the new DEM path;
- **Add a data entity as a leaf data entity:** this type of change doesn't affect any existing data entity paths, so it won't affect existing services but may require the addition of new services. There may be a need to create new services to deal with the new data structure. For example, consider a task NodeLayout (DEM_{Node}) which accepts DEM_{Node} as a parameter, and a service NodeLayout (DEM_{Node}). When DEM_{Node} evolves by the addition of a leaf data entity (such as a "Z" co-ordinate), the relationships back to tasks and services that use DEM_{Node} can be used to determine if a new NodeLayout service is required, in the form of a new concrete service NodeLayout (DEM_{Node}) which provides an implementation for 3D nodes. Of course, this will only work if assumptions about the structure of data in services can and have been extracted out as a parameter to the tasks and services. In effect, this is polymorphism on data structure.

The removal of a data entity can have ripple effects on a DEM path which uses the data entity. Adaptability can't be improved with respect to this.

6 Summary, Discussion and Conclusions

This chapter began with a description of existing types of data model, discussing their advantages and limitations with respect to evolveability and coming to the conclusion that many of the existing models, whilst productive for the task for which they were originally designed, have shortcomings with respect to evolveability. One of the aims of this thesis has been to determine the appropriate models and information contained within them that would produce inherently evolveable software. In this chapter, an approach to data modelling has been described that:

- Allows software to maintain an explicit model of its own data structures;
- Is generic enough to allow the representation of many different types of data structure;
- Allows the representation of data mappings, an important separation of concerns that is both not *explicitly* present in existing software languages, architectures and models and is a potential target for evolution;
- Provides a way for the software or software engineer to manipulate data for software evolution using well-defined and well-understood data evolution operators, whose interactions with other software entities in a software system (specifically FSEs) is well-understood.

The use of a generic data model provided by DEMs and DIMs allows many different data structures to be represented without implicit assumptions and constraints which would normally provide an "inertial" force to change (for example, representing data as a record when a linked list would be more appropriate). In addition, by adopting the most general data structure possible to model data, the graph, changes in data structures can be made with ease without affecting the representation. This is a subtle idea, probably best expressed in terms of an example. Imagine a data structure is expressed in terms of a record, but later needs to be expressed in terms of a linked list. The inflexibility of the representation of these data structures in traditional programming languages would make such a change difficult. In contrast, representing the initial record data structure as a graph would make the transition to a linked list data structure more simple, because the basic representation is the graph. The mapping between the two can be expressed in terms of their respective graphs, making the mapping easier to understand and not having to be concerned with low-level programming language implementation issues.

The DEM change type taxonomy described in this chapter has provided a complete set of change types for DEMs. Hursch provides a set of transformations for object-oriented data that are complete i.e. any class transformation can be expressed in terms of a sequence of them [Hursch95a]. However, these transformations (or mappings) are restricted to object-oriented models. The aim of this chapter has been to develop a similar classification which is tied to the most generic data structure possible, the DEM.

As has been recognised in this chapter, however, there are certain types of change to data that require changes to the capabilities of software. An example of this is when a 2-D graph DEM is changed to include a new z co-ordinate. This will then require a change to, for example, the graph layout algorithm. The next chapter describes such changes, along with other triggers for changes to the control aspect of software.

Finally, it is difficult to improve the adaptability of DIMs and DEMs with respect to changes in DIMs and DEMs on which they depend because of the inherent high dependencies which exist. It is also difficult to localise evolution of data entities so that data instances aren't affected. This is also due to the high dependency between data instance information and data entity information.

Chapter 7

Functional Software Entity Evolution and Evolveability

1 Introduction

This chapter explores functional software entity (FSE) evolution and evolveability. It begins by describing how FSEs evolve, using the evolution space approach described in chapter 4 and utilised in chapter 6. It then goes on to investigate the adaptability of FSEs with respect to other software entities, and the flexibility of FSE sub-models (an FSE sub-model is the part of the software entity model consisting of FSEs and their inter-relationships) in general.

Functional abstractions in traditional programming languages make certain assumptions about their environment:

- They assume that another functional abstraction which they use will always provide a particular functional capability and have a certain behaviour. For example, function F wants a sort capability with particular characteristics such as speed and space constraints and assumes function Bubblesort can provide it. So, it inserts a hard link in the form of a message to that function. If the function can no longer provide the same behaviour with the required constraints, then there will be ripple effects on F and any other functions which use Bubblesort and make similar assumptions. The problems are amplified by the fact that some characteristics of Bubblesort aren't part of the functional abstractions' interface. This will have unknown ripple effects on clients (or dependents) of the function. Hence, a change in the speed characteristics of Bubblesort may have ripple effects on its dependents, but this will not be explicitly apprarent because of the lack of an explicit aspect of the interface of Bubblesort which allows this to be modelled;
- Assumptions about the form and semantics of the FSE call. Programming languages typically assume local, synchronous FSE call semantics. Some languages assume all FSEs return a result, whilst others distinguish between procedures which return no result, and functions which do return a result. Other forms of FSE call must be built on top of the basic programming language abstractions;
- Assumptions about the form of their formal parameters. This includes the number and types of formal parameters.

The approach to overcoming these problems, which is described in this chapter, is based on a number of factors:

- Increasing the genericity of FSEs, thereby lessening the assumptions they make about their environment and improving their adaptability;
- Utilising reflection to model aspects of FSEs not traditionally modelled such as duration, their relation to the requirements which they implement etc.

By treating the functional aspects of software (the FSEs) as data, one can provide a reflective model which can be manipulated by evolution operators, much like data was in chapter 6. Early research on reflection was performed using LISP, which treats the computational aspects of the software[1] as data that can itself be processed by other computational elements. The basic data structure is the list and everything in LISP from code to data is treated as a list, as shown in Figure 1. This provides a shared or homogeneous mechanism for representing and thereby interpreting reflective information. The lack of many constraints on this representation also provides a generic mechanism for representing all the types of functional software entity that need to be modelled.

Often, there is a need to represent relationships between lists i.e. between LISP constructs. This is accomplished by embedding lists within lists to which they are related. The disadvantage of this form of representation is that these relationships are implicit, stemming from the fact that the best representation for reflective functional data (i.e. data about FSEs) is a graph-based model, which allows relationships between lists to be represented explicitly. This is depicted in Figure 2, in which the lists SE_{21} and SE_{22} are both related to SE_{11}. In the graph model, lists are represented as sibling relationships, and relationships between lists are represented as parent-child relationships.

Hence, the best, most generic type of model for both data and FSEs is a graph-based model which, in SEvEn, is the DEM (described in chapter 6) and the software entity model (described in chapter 5), respectively.

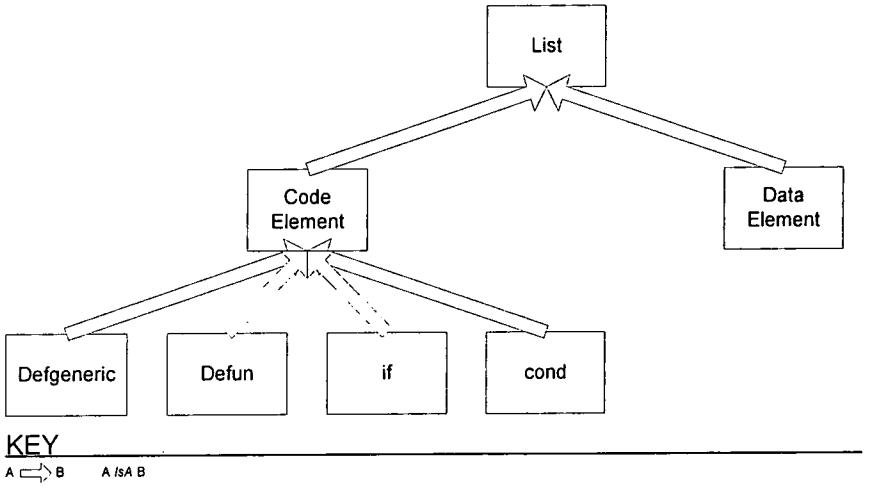


Figure 1 - Lists as the Unifying Representation in LISP

[1] Computational aspects include the following: procedure calls, iteration, conditionals, sequencing.

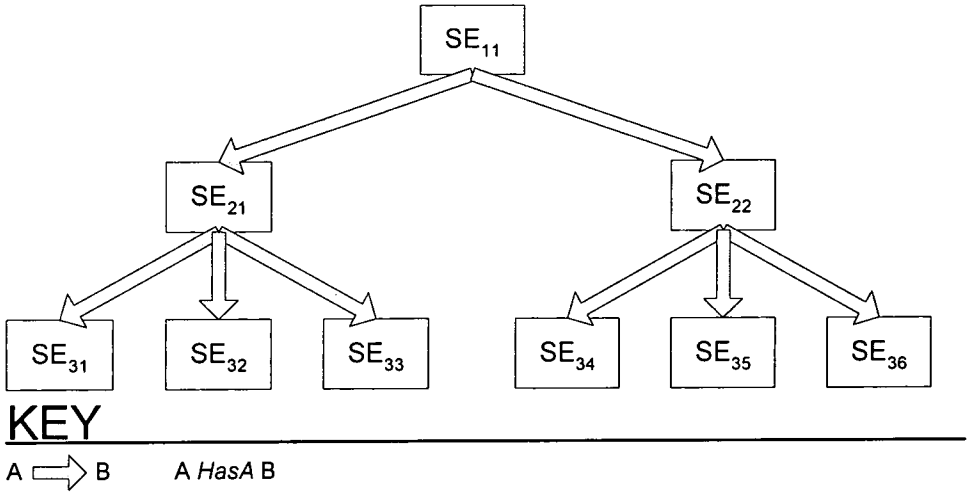


Figure 2 - Representing Relationships Between Lists

The inadequacy of existing functional abstractions is apparent when changes in functional abstractions produce unexpected side effects on their dependents. By adopting a richer functional abstraction model, which allows more characteristics to be modelled, the effects of change can be more easily determined and managed. These characteristics include:

- The task-service relationship;
- "Behaviour parameters";
- Types of evolution, the cross-product of the evolution operators with the elements of the functional abstraction.

This chapter describes the evolution spaces of the FSEs identified in chapter 5, and then goes on to discuss their evolveability.

2 FSE Evolution Spaces

2.1 Service Evolution Spaces

Kishimoto et al consider what they call "method adaptation" as a way of adapting object methods (which are related to SEvEn's services) to changes in their environment [Kishimoto95a]. Their research is concerned with method adaptation as a result of object migration and reuse, so that changes in the method environment are caused by the movement of the object to a new environment. However, what they term "method adaptation" is misleading since it is not the methods themselves that adapt (the methods still produce the same behaviour), but the messages that adapt. In comparison to SEvEn, the task interface stays the same, but the (brokered) mapping between the task and the service is changed, so that the same behaviour is observed[2] but a different set of services is called (possibly with different interfaces) in order to overcome the change in environment. In their work, Kishimoto et al base the mapping between task (what to do) and

[2] For some definition of behaviour that is based on the input-output relation, and specifically excludes non-functional (or similarly highly abstract functional requirements) aspects such as speed.

service (how to do it) using the concept of "producer intention", which corresponds closely with the notion of task in the SEvEn framework, and "method specification", which corresponds closely with the notion of service in the SEvEn framework. However, the mapping between the two is brokered through the use of a human user due to the intelligence required to map a producer intention to a method specification.

Types of service evolution which comprise the evolution space of a service are shown in Table 1. 1 and 2 require actual parameters in the form of DIMs and are thus constrained by the DIMs available in the process state space at the point in the call graph where the service is called. This classification circumscribes the evolution space of services into change types, but this alone is not enough for determining the effects of service evolution on other software entities. For this, the change types need to be linked to how they affect the characteristics and attributes of the service which together comprise the service interface (which was described in chapter 5 section 3.1.2.1). Hence, Table 1 also shows the effects of each type of service change on the service interface.

	Service Change Type	Effect on Service Behaviour	Comments
1	Adapt existing message	Behaviour-extending and/or Behaviour-conflicting	See section 2.3 on message evolution spaces.
2	Add new message	Behaviour-extending and/or Behaviour-conflicting	
3	Remove existing message	Behaviour-reducing	
4	Adapt existing formal parameter	Behaviour-preserving and/or Behaviour-extending and/or Behaviour-conflicting, dependant on type of adaptation	See chapter 6 on DIM and DEM evolution spaces.
5	Add formal parameter	None[3]	
6	Remove formal parameter	Behaviour-preserving and/or Behaviour-reducing	

Table 1 - Service Change Types and Effects on Service Interface

A simple taxonomy of the effects of service changes on behaviour is:

- Behaviour-preserving;
- Extension of behaviour, in which the new behaviour has no conflicts with existing behaviour;
- Replacement of behaviour, in which an existing aspect of the service's behaviour is changed;
- Removal of behaviour, in which an existing aspect of the service's behaviour is removed;
- Behaviour-conflicting, which means that two services produce behaviours which conflict[4]. This conflict arises because of a conflict in requirements, can occur for any number of different reasons, and is typically domain-dependant.

[3] Simply adding a parameter doesn't change the behaviour.

For data evolution, certain extensions of the data need not affect any dependent software entities because the new data structure is a generalisation of the previous data structure. Hence, the new data structure extends the modelling power of the old data structure so that it is able to model anything that the previous data structure was able to model. Behaviour is a similar characteristic for services as modelling power is for data. However, the characteristics of these two concepts with respect to evolution are very different. For example, extending the behaviour of a service may generalise the service with respect to some characteristic (such as speed of execution of the service), but this is problematic for any services using the service because the evolved service no longer provides exactly the same requirements as it did before evolution. Services use other services because they satisfy a particular set of behavioural requirements, which may be invalidated by changes to the service.

Behaviour extension is governed by the fact that it doesn't change the *existing* behaviour of the service. In other words, the requirements satisfied by the existing behaviour of the service don't conflict with the new requirements. Changes that affect the existing behaviour of a service do so because they:

- Conflict with assumptions made by the service;
- Affect data that is used by the service;
- Affect some external resource that is used by the service.

There are constraints on the evolution of services because they implement a task, which provides a boundary on the behaviour of the service. Hence, the task "sort" is implemented by a number of sort services, or algorithms, such as "BubbleSort". In reality, "BubbleSort" wouldn't evolve because it is an algorithm but, if it did change, the constraint imposed by the "Sort" task which it implements would rule out most changes to the service. This would include all changes that change the input-output relationship between the input and output data to the service.

2.2 Process Evolution Spaces

In chapter 5, process software entities were introduced as a process abstraction mechanism in software. For example, a data-oriented software system that filters the data passed through the system according to some specific rules would have a main thread of control to read in the data (possibly from a file), filter the data (possibly sorting it) and then output the data (possibly to a file).

The aspects of a process that can change are:

- The call graph, and;
- The state space (consisting of actual parameters for the messages in the call graph).

Applying the cross product of the evolution space operators add, remove and adapt and the two aspects of a process produces the evolution space shown in Table 2.

---

[4] This is also known as "feature interaction" [Zibman95a].

Evolution Space operator/Process Aspect	State Space	Call Graph
Add	Add Data Instance	Add New Message
Remove	Remove Data Instance	Remove Existing Message
Change	Change Data Instance (see chapter 6 section 4.2)	Change Existing Message (see section 2.3)

Table 2 - Process Evolution Space

These change types are mutually exclusive and form a complete change set with respect to processes in the sense that any process changes can be expressed in terms of a combination of the change types shown in Table 2. However, there are certain constraints on these change types. Adding a message to an existing process is dependant on the expressivity of the message (see section 2.2.1). Removing an existing message from a process may affect messages further on down the call graph, which use resources and DIMs that this message has produced or updated in some way.

2.2.1 Call Graphs and Message Expressivity

A major problem when integrating a new service or an evolved service into an existing process call graph is that of message expressivity; can the formal parameters of the service be mapped to the state space of the process? This is dependant on:

- The start state space of the process;
- The *Produces, Removes* and *Updates* relationships of all messages in the Call Path (see chapter 5 section 3.2.1.1) from the base message to the point of integration.

as the following definition expresses:

Definition: a message is expressible in terms of a state space if the data used by the message is part of the state space at that point in the call graph.

Expressivity is therefore a dynamic characteristic. That is, a message may be expressible at some points in the call graph but not expressible at other points, because of the dynamic nature of the data in a state space – data can be created and destroyed at run-time, as expressed by *Produces* and *Removes* relationships. In addition, evolution of a state space can affect message expressivity by affecting the actual parameters of a message.

Figure 3 shows the flowchart of an algorithm for determining the expressivity of a service, based on the observations above. The algorithm proceeds by first constructing a state space image, the state of the state space at the point of service integration into the call graph. This is accomplished by taking the start state space and, for each message in the Call Path before the point of integration, applying any *Produces*, *Removes* and *Updates* (i.e. any potentially destructive DIM operations) operations to the state space. The algorithm then determines if there is a potential mapping between each formal parameter of the service and a DIM in the state space image. The algorithm then considers any failed mappings

and determines, for each formal parameter without a mapping, whether there exists a DEM mapping which would allow a DIM in the state space image to be mapped to a DIM of the formal parameter.

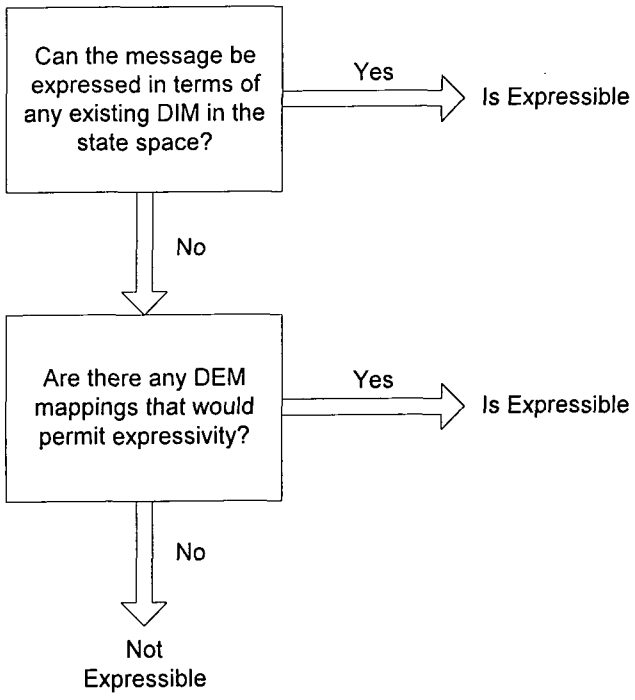


Figure 3 – An Algorithm for Determining Service Expressivity

The conclusion that a message is not expressible in terms of a process doesn't mean that the service cannot be called, just that one of the following must be performed: extra data must be introduced by a service instance that is called *before* the new service instance. This, in turn, requires either:

- The adaptation of an existing service in order to *produce* the required data;
- The integration of a new service with the process that *produces* the required data;
- The introduction of a new DEM mapping that permits the transformation of an existing DEM in the processes' state space to the required DEM.

2.3 Message Evolution Spaces

The evolution space of a message is governed by the evolution spaces of the software entities of which it is composed:

- The target task;
- A set of actual parameters.

as shown in chapter 5 figure 13. The evolution space of a message is shown in Table 3.

Evolution Operator/Evolution Target	Task	Actual Parameters
Add	N/A	These occur as a result of a change in the calling service.
Remove	N/A	
Adapt	See section 2.4.	

Table 3 - Message Evolution Space

2.4 Task Evolution Spaces

Applying the *evolution space principle* to tasks produces an evolution space consisting of the evolution types shown in Table 4.

Evolution Operator/ Evolution Target	Service	Implements Relationship
Add	The new service is related to the task through an *Implements* relationship.	This is not allowed, because *Implements* relationships may only be created in conjunction with service creation to link the service to an existing task.
Remove	Both the service and the *Implements* relationship are removed.	Not allowed because this would produce an inconsistent software entity model.
Adapt	The service evolves.	The parent of the *Implements* relationship (the service) is changed, thereby changing the mapping to another service.

Table 4 - Task Evolution Space

3 FSE Evolveability

3.1 FSE Flexibility

The aim is to break the direct dependence of services on other services, which is a characteristic of existing software languages and models. This is approached through the use of tasks and services, which separate out what to do from how to do it. Services are expressed in terms of messages, which encapsulate tasks, which provide a mapping to a particular service based on a set of pre-defined behaviour parameters. This mapping can be changed at run-time through changes in the behaviour parameters. The thesis doesn't describe how to do this in particular cases and domains, it only provides an architectural framework. It is unclear whether generic rules or heuristics exist for accomplishing this.

3.1.1 Binding Time and Glue-less Services

Delayed binding means that the onus on deciding the mapping between a task and a service lays more with the software and hence is based on increased modelling. Behaviour parameters provide a way of loosening the binding between what to do (the task) and how to do it (the service) by allowing a task to choose from a set of services based on the values of the behaviour parameters. This also has the advantage of providing increased extensibility of the architecture with respect to functionality. The onus is on the software engineer to determine the task abstractions which are relevant in the domain, the behaviour parameters which circumscribe each task abstraction, and the appropriate mapping from behaviour parameters to the service which implements the task with this particular set of behaviour parameters. The task abstractions model similarities in the domain, whereas behaviour parameters and services model the differences in these task abstractions. The difficulty lies in determining the behavioural similarities in a domain, and in integrating new behaviours with the existing task-service model.

"Glue" is an important characteristic of the *Calls* relationship which exists between FSEs. There are two main types of glue:

- **Special purpose**, which is generally hard-coded. For example, a sort program explicitly calling a sort service;
- **General purpose**, in which there is some late binding [ComponentGlossary]. A good example of this type of glue is polymorphism, in which the service called is only determined at run-time based on the type of object. Another example is that of Seiter's work, in which the next function to be called is determined by an increased context consisting of:

  - The class of sending FSE;
  - The object ID of sending FSE;
  - The task required;
  - A context object passed as a parameter to the task [Seiter98a].

  The difference Seiter's work makes to an object-oriented model is the provision of object-level forwarding where the target of a message is based on the class of the message and the sender of the message (as shown above), as opposed to traditional object-oriented models in which the target of a message is based on just the class of the message. This provides a richer modelling framework and creates a more parameterised method-call space. Similarly, the task-service abstraction provides a way to further delay the binding between what to do and how to do it.

However, any new service can't simply be slotted into an existing running software system and be expected to immediately take part in the software system. The promise of glue-less components has been this very notion, where context independent components are slotted in and expected to work with the existing system. Any new service will have some relationship with the existing software. In order for the notion of glue-less components to work, this relationship must be recognised and appropriately modelled. Typically, this relationship is made explicit by glue, so that the link between a new sort service and the existing software system, for example, is hard-coded in by explicit message

sends or procedure calls. However, by appropriately modelling new services, there is no need for explicit coding of glue components like this. For example, a new "BubbleSort" service can be expressed in terms of an existing task abstraction. The extensibility this provides allows the new service to simply be slotted into the existing model and take part in the running software system. It is difficult, however, to deal with new services which can't be expressed in terms of any existing task abstraction.

3.2 Service Evolveability

3.2.1 Adaptability With Respect to DEM Evolution

There are two aspects to service adaptability:

1. Adaptability with respect to changes in the interfaces of services which are used;
2. Adaptability with respect to changes in the formal parameters of the service.

1 is discussed in section 3.2.2. Of the total number of dependencies between software entities in any software system, one of the most common dependencies is the one that exists between services and DEMs, point 2 above. As stated in chapter 5 section 3.1.2, services can only access data through their parameters. Hence, adaptation of services to data is equivalent to adaptation with respect to parameters. The choice of representation between these two types of software entity can take one of the forms shown in Figure 4 (b) and (c), which are types of (a).

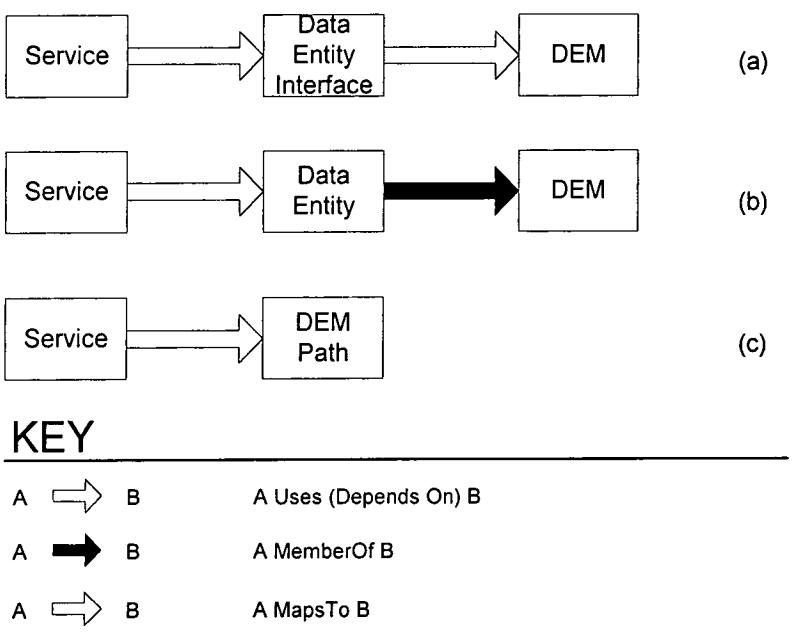


Figure 4 - Service Dependence on Data

In Figure 4 (b), the dependence is on the individual data entity. In Figure 4 (c), the dependence is on the data entity as represented by a DEM path. Changes in the structure of the data are handled in different ways. In (b), the mapping between the service and the data entity stays the same, even though changes in the DEM may affect the relationship between the data entity and other data entities in the DEM. In (c), changes in the DEM will affect the DEM path and also

the service, because the structure of the DEM is visible to both the DEM path and the service. (b) is hence the better approach, because it introduces an interface between the service and the data entity which successfully hides the structure of the DEM. Hence, particular types of change (those that don't break the data requirements of the service) are localised to the DEM and ripple effects thereby reduced.

Adaptability with respect to DEM evolution is governed by:

- The semantics of the DEM doesn't change i.e. requirements which the service makes of the DEM don't change;
- The implementation of the DEM changes.

Changes in implementation then consist of:

- Changes in the structure of the DEM e.g. change from an array to a linked list;
- Generalisation of the DEM i.e. the new DEM is a generalisation of the pre-evolution DEM (see chapter 6 section 3.3).

Consider the use of DEMs as data models in a sort program consisting of three FSEs (see Figure 5).

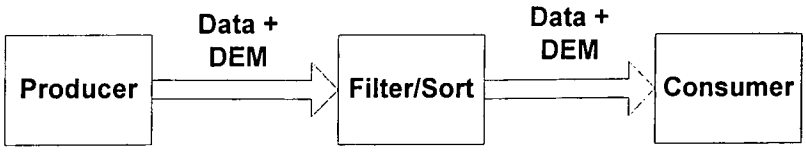


Figure 5 - Sort Program Consisting of Three FSEs

Figure 6 – Desired Data Flow for BubbleSort

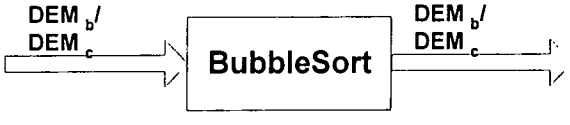


Figure 7 - Actual Data Flow for BubbleSort

The Filter/Sort FSE expects data of a "type" (i.e. conforming to a particular DEM) for its BubbleSort algorithm, but in a particular case receives data of a type (i.e. using a DEM) with which it is not familiar (see Figure 6 and Figure 7, and Figure 8, Figure 9, and Figure 10 for the DEMs). What does the sort FSE do? It could simply give up and ask the software maintainer to change it so that it can perform its sort on the new data. This, however, is no advancement on current techniques. Another approach is for the sort FSE to use a specially-written specification of the desired mapping between any data of a particular type to any data of another type. A broker agent would hold such specifications,

allowing any FSEs in the system to ask the broker agent if there is an existing mapping between two arbitrary DEMs. However, this is also not desirable. Imagine that the sort FSE expects data of "type" DEM_a (Figure 8), but instead receives data of "type" DEM_b (Figure 9). It would not be desirable to convert DEM_b to DEM_a because information would be lost in such an operation. This in itself is not problematic since information loss in certain mappings may be desirable when that lost information is not required by the next FSE to use the data, but in this case the lost information is important and therefore another way must be found.

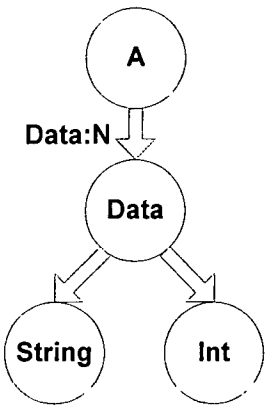


Figure 8 - DEM_a

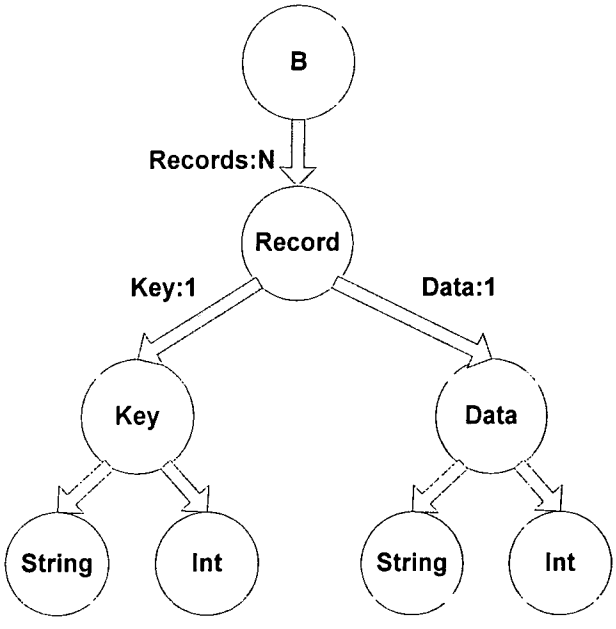


Figure 9 - DEM_b

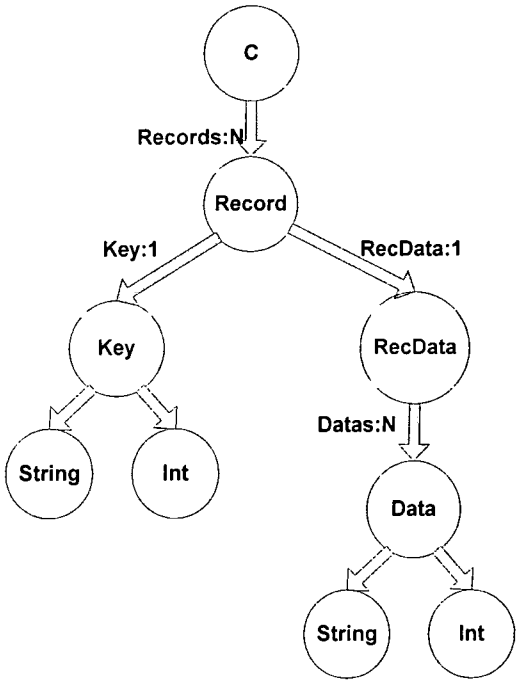


Figure 10 - DEM_c

Now let us look at the BubbleSort code (see Figure 11). Notice that the code is written in terms of DEM_a. If data of "type" DEM_b arrives at the sort service, then the sort service needs to ascertain whether the existing sort algorithm is valid. There are two aspects to this validity:

1. **DEM-validity :** validity with respect to the new DEM;
2. **Requirements-validity :** validity with respect to the software requirements.

```
do {
        Swapped = false;
        for (int i = 0; i < NumInstances (A 0,Data) - 1; i++) {          (a)
                if (Compare ([A 0,Data i], [A 0,Data (i+1)]) > 0) {      (b)
                        Swap ((A 0,Data i), (A 0,Data (i+1)));           (c)
                        Swapped = true;
                }
        }
} while (Swapped);
```

Figure 11 - $BubbleSort_{DEM\,a}$

DEM-validity refers to the compatibility between DIM paths and DEMs, specifically whether the data entities and cardinalities used in a DIM path are valid with respect to the DEM. Software requirements validity provides a semantic

aspect to this, requiring that the data entities used in the DIM path being considered are semantically-related[5] to the new DEM.

```
do {
      Swapped = false;
    for (int i = 0; i < NumInstances (B 0,Record) - 1; i++) {
        if (Compare ([B 0,Record i,Key 0], [B 0,Record (i+1),Key 0]) > 0) {
            Swap ((B 0,Record i), (B 0,Record (i+1)));
                                        Swapped = true;
                }
          }
} while (Swapped);
```

Figure 12 - $\text{BubbleSort}_{\text{DEM b}}$

One can analyse these two forms of validity with respect to the sort program as follows. Firstly, check the DEM-validity for $\text{BubbleSort}_{\text{DEM a}}$ with respect to DEM_b. The approach is to check each data-use statement in the sort code. For example, line (b) Figure 11 uses the "compare" service:

$$\text{Compare } ([ED_1\ 1, ED_2\ i], [ED_1\ 1, ED_2\ (i+1)])$$

The ED_i refer to data entities in DEM_a, and may not correspond to semantically-related data entities in the "target" DEM_b. In this case, the ED_i used by the compare service are semantically-related in both DEM_a and DEM_b, but the use of the DIM paths here is not DEM valid since the DIM paths refer to non-primitive data entities and compare expects primitive data entities as its two parameters, as shown in Figure 13 (where $DEM_{input} = DEM_f$ and $DEM_{output} = DEM_g$). In other words, the validity can be checked in terms of actual parameters and formal parameters. The code is DEM-valid with respect to the new DEM if the actual parameters match the formal parameters, and isn't valid otherwise.

[5] The term "semantically-related" means entities in one DEM that are the same in another DEM. For example, a record consisting of a key and N data items in two different DEMs.

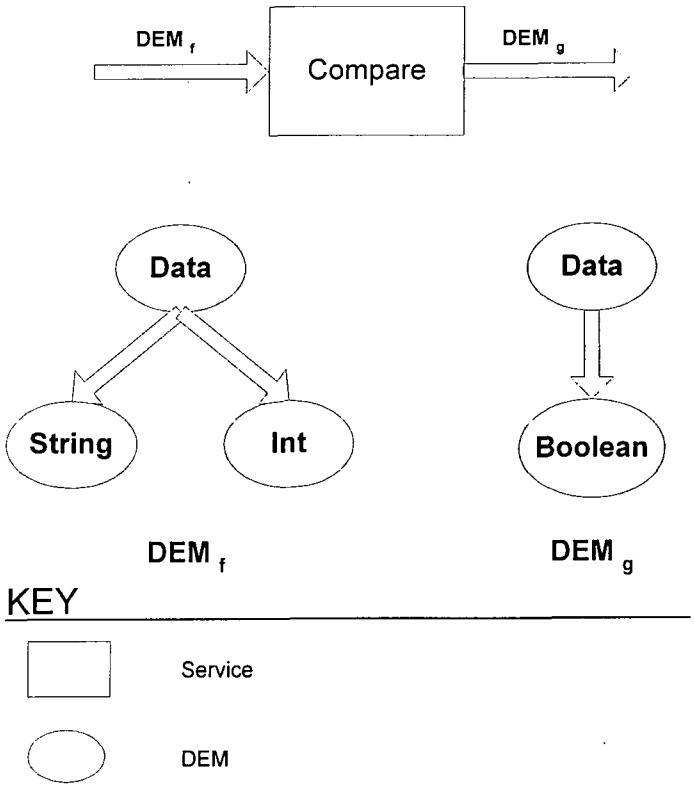


Figure 13 - Data Flow for Compare

A way around this would be for the software maintainer to provide a mapping between the DIM paths used in the existing code to the DIM paths that will provide requirements validity again. In the example above, in order that $\text{BubbleSort}_{DEM_a}$ can be used on data of "type" DEM_b, line (a) can be mapped as follows:

$$\text{compare}([ED_1\ 1, ED_2\ i], [ED_1\ 1, ED_2\ (i+1)])$$

$$\Downarrow$$

$$\text{compare}([ED_1\ 1, ED_2\ i, ED_3\ 1], [ED_1\ 1, ED_2\ (i+1), ED_3\ 1])$$

i.e. an extra part "ED_3 1" has been appended to each DIM path. This addition makes the code requirements-valid with respect to DEM_b.

It is clear and quite straightforward that a set of operations can be devised which, when applied to one DIM path, will produce another valid DIM path. The validity requirements here are those of DEM-validity i.e. that the new DIM path is valid with respect to a particular DEM. The operations are:

1. **Append:** DPCs (DIM Path Components) are appended to the existing DIM path in such a way that they preserve the validity of the DIM path with respect to the DEM;
2. **Modify:** where existing cardinalities in the DIM path are changed so as to be consistent with the cardinalities in the DEM. For example, it would be invalid to change "ED_k 1" to "ED_k 5" if the DEM stated that the maximum cardinality for ED_k was 4;

3. **Remove:** removal of a portion of the DIM path starting from a certain point and ending at the *end of the DIM path.*

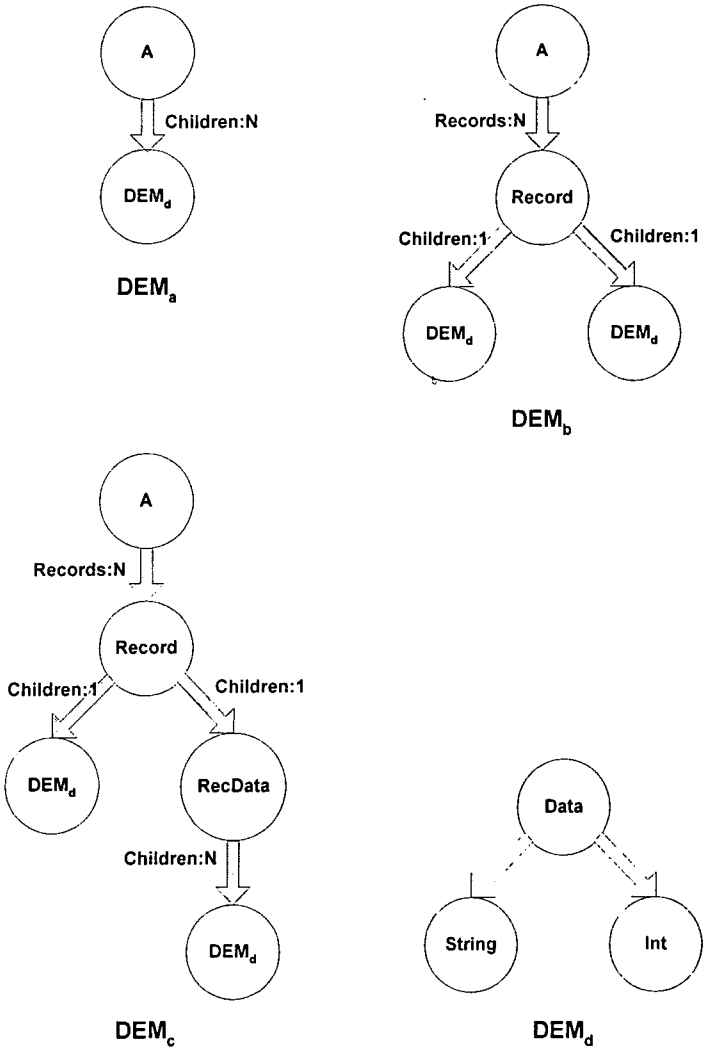


Figure 14 - Reuse in DEMs

So, a general approach for changing code so that it can use a different DEM is as follows:

1. Map the code's DEM to the new DEM;
2. Alter the DIM paths used in the code using the operations defined above, using information provided by the mapping.

For step 1, notice the relationship between DEM_a and DEM_b, shown in Figure 14. Each contains a common DEM, DEM_d, that can be used to provide a mapping from DEM_a to DEM_b, as shown in Figure 15. Once this has been accomplished, the DIM paths can be updated using information from the mapping.

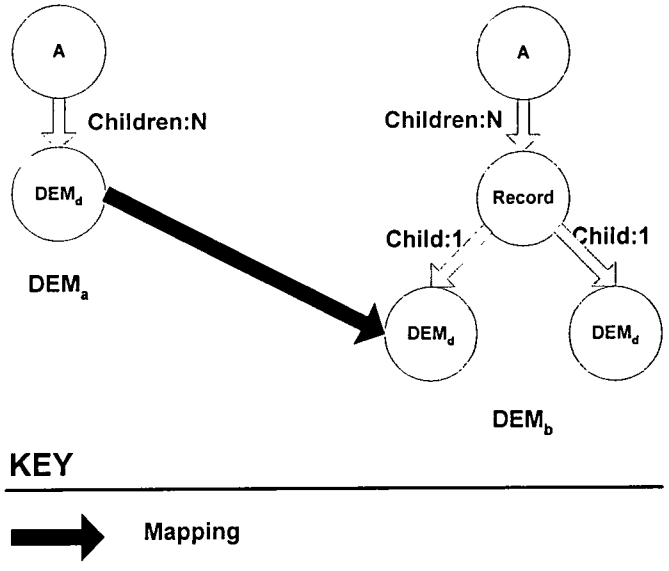


Figure 15 - Mapping DEM_a to DEM_b

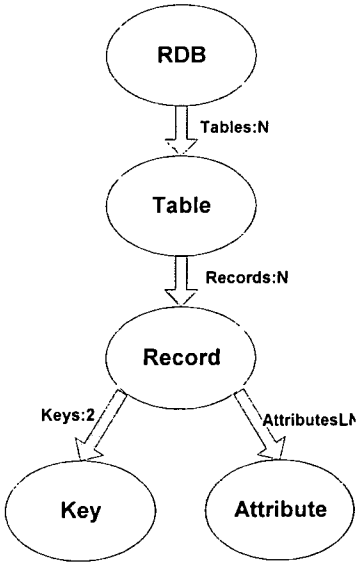


Figure 16 - DEM_{RDB}

Look at the relationship between the three DEMs concerned in the type of evolution being considered here: the input DEM (the DEM or set of DEMs which are the services' formal parameters) DEM_{input}, and the output DEM, DEM_{output}, and the actual data DEM, $DEM_{actual\text{-}input}$, which may be different than DEM_{input}. The data output from the software entity may also change, and is given by $DEM_{actual\text{-}output}$. For example, imagine that DEM_{output} is the Sort DEM in Figure 17 and $DEM_{actual\text{-}input}$ is the DEM_{RDB} in Figure 16. After sorting the data in DIM_{RDB}, the sort component couldn't output the data in the form of a DIM_{Sort}, because information would be lost. The data would have to be output as DIM_{RDB}. In effect, a change in the input data of one service percolates through the software.

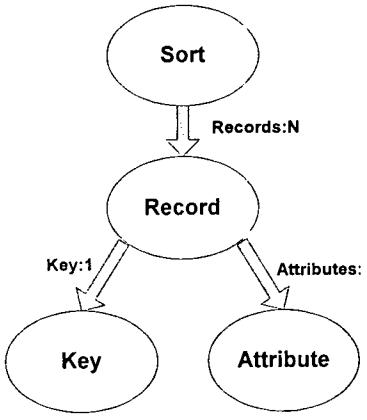


Figure 17 - DEM_{Sort}

Another problem arises when a service assumes a particular cardinality on a *HasA* relationship between two data entities. Take Figure 18 for example. Line (b) is where the keys of adjacent records are compared and in this case, the service assumes that there is only one key. When data arrives to be sorted and contains more than one key, as in DEM_{RDB}, the DIM paths (a) to (c) can be easily converted as per the protocol, but the code also needs to change. A solution to this is always to assume that the cardinality is N, at the expense of some inefficiency. This is shown in Figure 19.

```
do {
      Swapped = false;
   for (int i = 0; i < NumInstances (Sort 0,Record) - 1; i++) {                    (a)
      if (Compare ([Sort 0,Record i,Key 1], [Sort 0,Record (i+1),Key 0]) > 0) {        (b)
         Swap ((Sort 0,Record i), (Sort 0,Record (i+1)));                          (c)
                                    Swapped = true;
                }
         }
} while (Swapped);
```

Figure 18 - Bubble Sort Service

```
do {
    Swapped = false;
  for (int i = 0; i < NumInstances (Sort 0,Record) - 1; i++) {
    for (int j = 0; j < NumInstances (Sort 0,Record i,Key j); j++) {
        if (Compare ([Sort 0,Record i,Key j], [Sort 0,Record (i+1),Key j]) < 0)
        break;

      if (Compare ([Sort 0,Record i,Key j], [Sort 0,Record (i+1),Key j]) > 0) {
        Swap ((Sort 0,Record i), (Sort 0,Record (i+1)));
                        Swapped = true;
                  break;
                }
      }
    }
} while (Swapped);
```

Figure 19 - Revised Bubble Sort Service

In the protocol on p222, DEM_{input} is mapped to $DEM_{actual\text{-}input}$ and information from this mapping is then used to change the DIM paths in the service. The ideal situation is where $DEM_{actual\text{-}input} = DEM_{input}$ because then no conversion (or evolution) has to take place to a service before it can be executed.

The possible relationships between DEM_{input}, $DEM_{actual\text{-}input}$ and DEM_{output} are as follows ("=" indicates "the same as", "≠" indicates "not the same as"):

1. $DEM_{actual\text{-}input} = DEM_{input}$:

   (a)$DEM_{input} \neq DEM_{output}$ e.g. output = count of number of records in input;

   (b) $DEM_{input} = DEM_{output}$ e.g. filter/sort routine $\Rightarrow$ $DEM_{actual\text{-}output} = DEM_{actual\text{-}input}$;

2. $DEM_{actual\text{-}input} \neq DEM_{input}$:

   (a) $DEM_{input} = DEM_{output}$;

   (b) $DEM_{input} \neq DEM_{output}$;

In case 2(a), the change will percolate through this FSE because of the dependency between the output and input DEMs.

A major problem to be overcome is when DEM evolution alters the domain. A classic example of this is when $DEM_{2Dgraph}$ is changed to a different domain, $DEM_{3Dgraph}$, by the addition of a new co-ordinate. The relationship between a DEM and its domain is a semantic one and difficult to model. It first requires a definition of the term "domain"; what are the characteristics of a domain which classify it as a particular domain such as "2Dgraph" or "Sort"? Does it include

data and function and, if so, which aspect or characteristics of data and function? Does it include software architecture, since some domains are intimately coupled to a particular software architecture? One conclusion is that there is no generic set of characteristics that defines a domain, because each domain can be characterised by a different set of characteristics.

Another example is evolution of a DEM representing a singly-linked list to a DEM representing a doubly-linked list. This shouldn't affect any dependant services because they don't depend on the structure of the data entity "Node":

- GetNext (...) depends only on Node.Next;
- Iterate (...) depends on getNext (...) and so, by implication, depends only on Node.Next

However, evolution of $DEM_{2DGraph}$ to $DEM_{3DGraph}$ will affect a graph layout service because the service "Distance (Node1, Node2)" which it uses is affected by the change because it depends on the *structure* of Node. This can be determined from the entity path passed to the "Distance (Node1, Node2)" service and the fact that "Distance (Node1, Node2)" depends on a particular structure of "Node". Any change to "Node" will invalidate "Distance (Node1, Node2)". Of course, this need not always be the case; the dependence of a service on a data entity is a semantic one which may or may not be broken by evolution of the data entity.

What can be added to the basic DEM in order to allow particular data type changes to trigger a behaviour change, or show that a behaviour change is necessary? In $DEM_{2DGraph}$, the "Node" entity has two child entities of type "Coordinate". Adding another child entity of type "Coordinate" invalidates a graph layout service as discussed above because the requirements are no longer met, not because any existing entity paths are no longer valid. This can be overcome by an improvement to $DEM_{2DGraph}$ that either prevents a new co-ordinate from being added, or recovers from the addition of a new co-ordinate by calling a user-specified service. This user-specified service is triggered by a pre-condition which triggers the service if a new coordinate is added. In some cases, the addition of a new data entity to a DEM need not cause ripple effects on any services using it. This depends on the leeway afforded the DEM by the requirements which are placed upon it by any services using it. For example, a sort service may require:

1. The data to be sorted to consist of an array (or sequence) of data entities;
2. A "Compare" service which compares any data elements in 1.

This is really a problem for the design phase of software development, but has consequences for evolution. It emphasizes the dependence of ease of evolution on the earlier stages of software development, in this case design, because evolution allows essentially unconstrained changes to the data without regard for what effects this might have on other parts of the system. In this case, evolution results in the addition of another coordinate and without the constraint this evolution would produce software that fails to match its requirements. Hence, the onus is on the software developers to analyse the DEM and determine the scope of changes allowed and classify them as either:

- Changes that don't affect the requirements;
- Changes that do affect the requirements.

In the latter case, of which the $DEM_{2Dgraph}$ change above is an example, measures must be built in to at least alert the software and maintainer that further evolution needs to occur. Note that these further changes are not really ripple effects in the traditional sense because ripple effects are more concerned with stabilising the software after a change, to ensure that the interfaces are correct etc.

3.2.1.1 Data Access Services and Interfaces

The idea here is to increase the adaptability of services with respect to DEMs by limiting DEM access to a small number of data access services, which together form a data access interface. Other services then use these services. In this way, changes to the data are limited to this small number of data access services. The problem with this approach is to limit the number of data access services so that changes to the data don't cause many ripple effects. Any changes that occur to a DEM will involve changing the relationships between data entities. For example, consider the DEM change depicted in Figure 20. Any data accesses of the form Document.Paragraph[i] will be invalidated by the change. Such changes can be encapsulated within a set of data access services, one for particular data entity pairs.

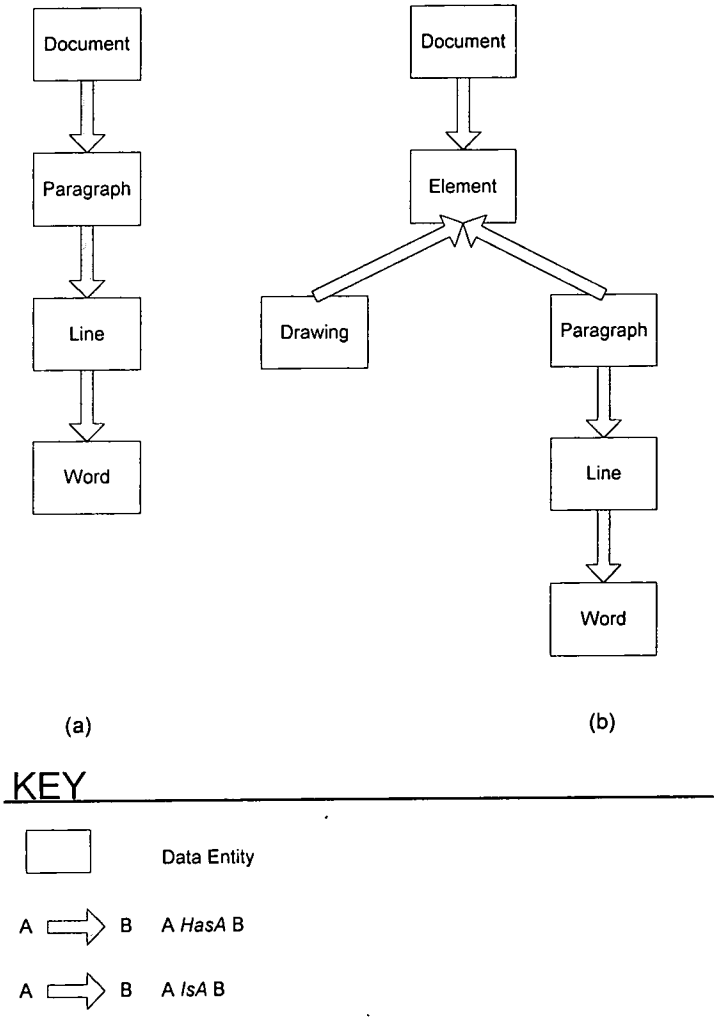


Figure 20 - DEM Change

Hence, we have:

- getParagraph (Document, int Index)
- getLine (Paragraph, int Index)
- getWord (Line, int Index)

These three services will be the only services affected by a change in $DEM_{Document}$. This is similar in principle to the use of abstraction and interfaces for services. Instead of having the situation shown in Figure 21 (a), where X_2, X_3 and X_4 perform particular tasks, these three services are abstracted out and placed behind an interface provided by service X_5, as shown in (b). If changes do occur to the implementation of X_2, X_3 and X_4, then X_1 is not affected (because X_5 hides changes in X_2, X_3 and X_4 only for changes that don't conflict with (explicit or known and implicit or as yet unknown) requirements of X_1). In (a), however, this would not be the case.

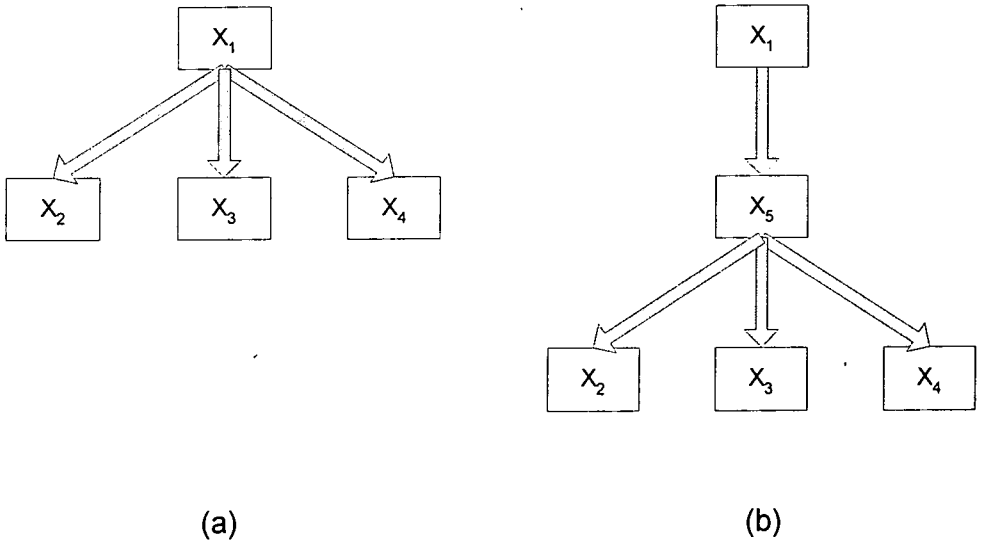


Figure 21 - Service Abstraction and Interfaces

The use of increased modularity as a mechanism to localise evolution can be extended to increasing the modularity of services through the identification of types of services, as shown in Table 5. This has the advantage that changes in implementation (of a service or DEM) affect only the specialised services. Hence, for iteration services, changes in implementation of the underlying data structure, for example from an array implementation to a linked-list representation, are contained.

Service Type	Description
Data Access	Services that directly access data on behalf of other services.
Iteration Services	Services which provide iteration over a data structure such as an array of linked list.
DEM Mapping	A DEM Mapping is essentially a DEM, but requires a special DEM mapping service to perform the mapping.

Table 5 - Service Types

Data accesses in a programming language are generally of the form:

X.a.b

where "X" is a record or an object, "a" is a data member of "X" whose type may be a record or an object, and "b" is a data member of "a" whose type may also be a record or an object. However, such data accesses hard-code data paths, which may break when the DEM evolves. The data access operator, "." in this case, needs to be replaced in order to improve the adaptability of data accesses with respect to DEM evolution. This can be approached in one of two ways:

1. Alter the underlying compiler by changing the behaviour of the "." operator to deal with "rubbery" *HasA* relationships between data entities;
2. Provide data access services at the user code level rather than at the compiler level, using a tool which parses a DEM in order to produce a set of data access services, as shown in Figure 22. These data access services can be used by user code instead of the "." operator. This tool has to deal with how changes in a DEM affect the data access services, which it does by taking as input the DEM before and after evolution.

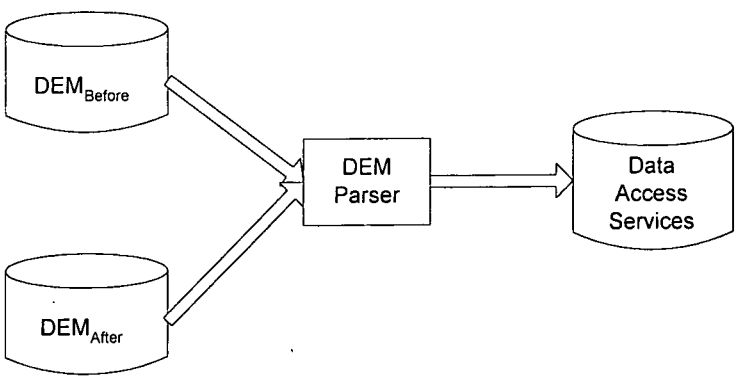


Figure 22 - DEM Parser and Data Access Services

The second option is the preferred option because it allows the advantages of data access services to be applied to traditional programming languages. The approach is to provide an API through which data accesses are brokered. This API consists of a single service:

getData (DIMPath DIMP)

to which is passed a DIM Path, as shown. This service walks the DIM model using the DIMPath and returns the appropriate data instance. It provides an interface which hides structural changes in the DEM referenced by the DIMPath.

In order to prevent these particular types of change from affecting services, the adaptability of DIM paths needs to include adaptability with respect to these change types. An analysis of how changes in a DEM can affect a DIM path produces the following conclusion: since data instances are accessed through a DIM path by walking the DIM from the

base instance towards a leaf data instance, any changes in the relationship between data entities could affect a DIM Path. Specifically, data instance access is based on data instance pairs ((DE_1, i), (DE_2, j)) such that DE_2 is a descendant of DE_1. Changes in the *structure* of a DEM can affect individual data instance pairs. This is overcome by making data instance pairs as general as possible. The getData (...) service does this by a traversal algorithm that tries to reach data instance (DE_2, j) given data instance (DE_1, i) by walking the DIM, an approach which is similar to that of the Demeter Project [Lieberherr96a]. In this way, no specific hard-coded structure is built-in to the getData (...) service. However, unlike Demeter which provides similar adaptability only for multi-class functions with respect to data, this approach allows a more fine-grained relationship between services and data. Hence, the use of DIM Paths in order to access data have the following advantages:

- They are not tied to an object-oriented model;
- They are finer-grained. Propagation patterns are multi-class behaviour entities which are able to adapt only to changes in an underlying class model. DIM Paths are finer-grained;
- They are not as complex as propagation patterns, but provide the same advantages with respect to adaptability as propagation patterns do.

In addition, DIM Paths can cope with the addition of new data entities to a DEM (especially when they affect existing DIM Paths in a DIM because of the looser *HasA* relationships between data entities). The disadvantages are that DIM Paths can't cope with:

1. Removal of data entities from a DEM;
2. Changes in the identity of data entities in a DEM.

but these are semantic considerations which need to be mediated by the software engineer anyway. Changes of type 1, for example, are probably a result of requirements conflict, which is out of the scope of this thesis.

3.2.2 Adaptability With Respect to Service Evolution

A large majority of the relationships in any software system will be *Calls* relationships between FSEs, such that a service X *Calls* service Y indirectly via Task T. The *Calls* relationship in current software languages is typically implicit (for example, function or method calls) and encapsulates very little semantic information, especially information regarding the nature of the contract between two services. The main component of this contract is the interface which the server service provides to the client service. Current software languages possess limited functional abstraction interfaces which basically consist of data parameters and, in particular, no modelling of which requirements the functional abstraction implements, such as behaviour and non-functional characteristics like duration.

Crelier discusses the effects of changing a module (or function) interface on clients (or dependants) of the module [Crelier95a]. However, the approach is limited for two reasons:

- It considers only changes in module *interfaces*, and;
- It only deals with the *extending* of module interfaces, by the addition of objects to the interface of a module. In addition, he concludes that the pure extensions to a module's interface shouldn't affect any clients of the module. This is only true in a syntactic sense because client modules won't need to be re-compiled. However, semantically this is an invalid conclusion, as was shown for additions to DEMs in the previous section. In addition, the addition of objects to a module interface may indicate a change in behaviour of the module, which can only be fully utilised by the client adapting to use these new objects.

One aspect of this contract in particular, which is not modelled, includes the requirements that the client makes of the server. Two characteristics of this contract are assumed:

1. The requirements of the client are completely met;
2. The requirements of the client are rigid and any change in the server's behaviour will invalidate them, thereby causing implicit ripple effects which may not be recognisable by looking at the code or design documentation.

2 causes problems for evolution of the server, since it is unknown which types of change in the server should affect the client and which changes in the server the client should be able to adapt to automatically. The reason for this is that requirements are typically under-specified, meaning that they leave aspects unspecified which later become important. For example, in a sort program, the client may simply require of the sort component that the data be sorted, without specifying the desired duration of the sort or any memory usage requirements that need to be met. If the sort algorithm needs to evolve (granted, probably unlikely), it will be unclear whether these changes affect the client's requirements of the sort algorithm. The changes may, for example, decrease the speed of the algorithm and hence increase its duration, which may affect the implicit duration requirements of the client or, indeed, some other function or sub-system higher up in the call graph. A solution to this requires that all present and future requirements for such service-to-service contracts be represented along with the effects of changes in the server on these requirements. This, of course, is not possible because it would require a predictive capability on the part of the software engineer. Hence, the following types of change in server behaviour can potentially have a ripple effect on any clients of the server:

- Changes in the server's behaviour which invalidate requirements that the client expects of the server;
- Changes in the server's behaviour which invalidate the client's duration requirement;
- Changes in the server's behaviour which invalidate the client's memory usage requirement (or some other software entity which imposes such a requirement on either groups of services or a software system as a whole).

Consider the following example, based on the simple mobile telephone system shown in Figure 23, in which the mobile telephone and telephone network are represented by state machines. In this example, a mobile telephone is designed at a particular stage in the development of the mobile telephone network's design. Hence, there is a dependency:

$$\text{Mobile Telephone}_i \; \textit{DependsOn} \; \text{Mobile Telephone Network}_j$$

Now, consider an upgrade of the mobile telephone network to version "k" (for k > j), whilst keeping the same mobile telephone version, "i" (where i, j and k represent particular stages in the development of the designs of the respective software subsystems):

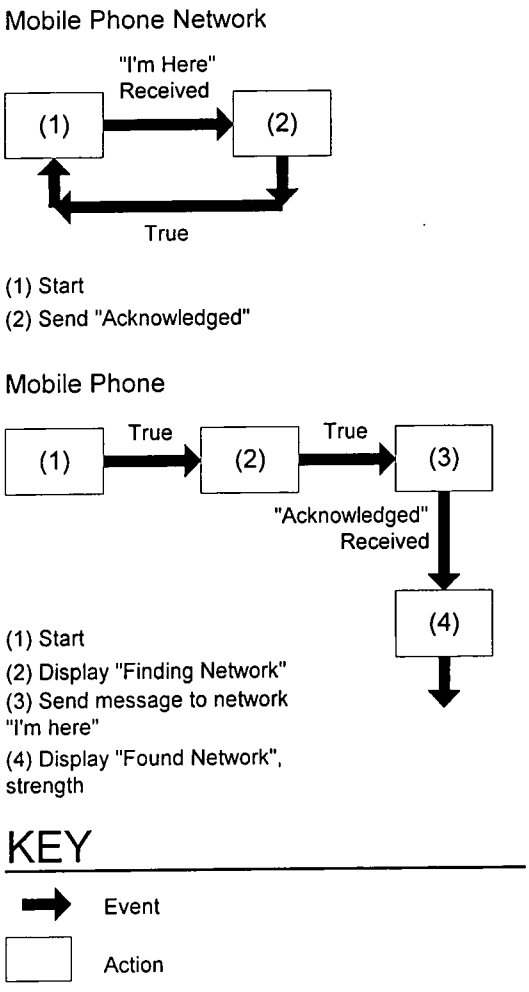


Figure 23 - A Simple Mobile Phone System

Mobile Telephone$_i$ *DependsOn* Mobile Telephone Network$_k$

The mobile telephone network design has changed, but the "old" mobile telephone design depends on the old version, j. The following are therefore true:

- The client's (in this case, the mobile telephone) requirements don't change i.e. the requirements which the mobile phone makes of the mobile phone network don't change;
- The implementation of the server (in this case, the mobile telephone network) does change i.e. the implementation of the mobile phone network does change. The new implementation may add new services or re-implement existing services. These changes may impinge on the interface between the mobile phone and the network.

Hence, the new mobile phone network (k) must still provide the same interface and behaviour to the mobile phone as before. A form of adaptability which is important here is that of ignoring events or messages. If this is adopted, the old

mobile telephone can still function even in the presence of a new mobile telephone network design which consists of a communication protocol which produces more events/messages. Hence, requests of the network must return data that is a generalisation (i.e. a more general data type or the same data type) of the old data.

However, an important aspect that must be considered is how the network copes with the old design of the mobile phone, which may not respond to new messages which it produces. In this case, there is a two-way client-server relationship: both software entities are clients and servers.

Chapter 6 section 4.2 discussed DEM generalisation and how a generalised DEM can be used in place of the original DEM with no effects on clients using the DEM. Applying a similar idea to services, however, causes problems because of the high coupling between the behaviour of a service and the requirements it implements. Given completely unconstrained requirements on a service, any change in the service would produce a valid generalisation. However, requirements are typically very constrained, which makes it difficult to generalise (or extend the behaviour of) services.

3.2.3 Adaptability With Respect to Message Evolution

Services are related to messages in two ways:

1. Services *send* messages to other services;
2. Services *receive* messages from other services.

3.2.3.1 Adaptability With Respect to Messages Received

There are three steps to be performed upon receipt of a message:

1. Parse, converting the information in the message into a software entity model;
2. Interpret, relating the model constructed in step 1 to the existing software system entity model. In more detail, this involves mapping the behaviour and data parameters to a service;
3. Execute/Perform.

as shown in Figure 24.

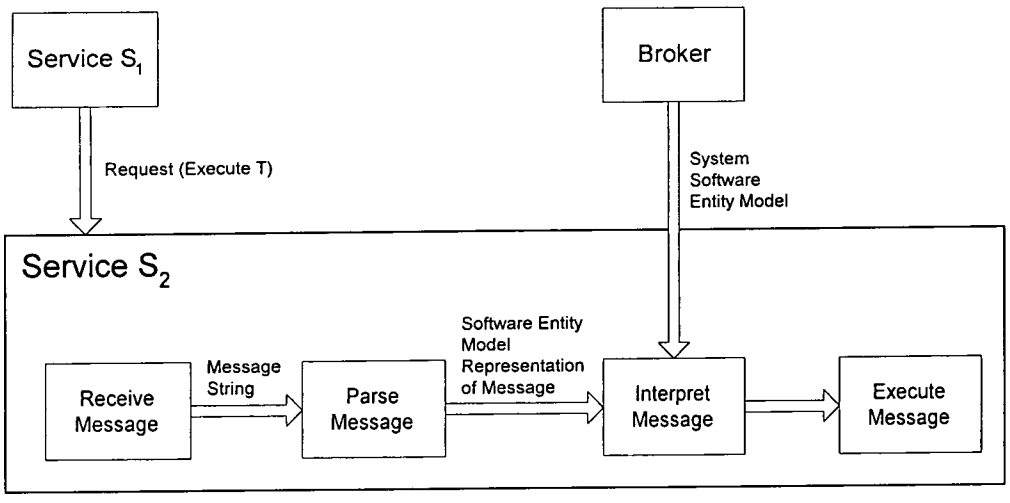


Figure 24 - Parsing, Interpretation and Execution of a Message

A typical problem with current software models is that the task information is hard-coded in a dependency created between functions, components and classes. As an example, imagine service S_1 wishes a task T to be performed (see Figure 24), which is directed to a call to service S_2. S_1 must specify what it wants S_2 to do in terms of concepts known to both services in order to allow parsing and interpretation. Messages have a pre-defined structure or syntax which all services can parse. A problem with traditional programming languages, models and architectures is that changes to actual parameters may render them un-parse-able by a function. In comparison, the use of DIMs and DEMs means that changes in an actual parameter (represented in terms of a DIM) will not render the changed actual parameter un-parse-able, but the message as a whole may become un-interpretable because there is no way of relating it to the software system's reflective software entity model. Hence, the message cannot be executed because:

- The software system isn't able to interpret the message;
- By implication, inability to interpret implies lack of an appropriate aspect of the software entity model, which in turn implies either a lack of capability to perform the message's request or a semantic heterogeneity problem.

This must inevitably result in evolution (either a new functional capability or new semantic information). For example, a service receives a message to perform an "X" layout operation on a graph. The service isn't able to interpret this message, so the software system must evolve to be able to interpret and execute the message. This will inevitably require a software engineer to write a service and incorporate this into the software system's software entity model (helped by a flexible software model like SEvEn). If, on the other hand, a service receives a message to display graph nodes as squares instead of circles, and assuming that:

- An appropriate service exists, and;
- The software entity model relates the message to this service through the broker;

then no user-involved evolution is required. In this case, evolution of the mappings between the two domains involved occurs.

This method of expressing the behaviour of a software system (that is, the use of a service request approach whereby a service requests another service to perform some task on its behalf) typically falls down when what is required can't be specified in terms of concepts known to the server service of which the request is being made. This can be helped by two characteristics of the domain:

- There exists a set of basic concepts;
- All existing and future requirements are expressible in terms of these basic concepts.

In this way, all new requirements can be specified to the process, which can then use the rules and constraints inherent in these basic concepts to configure the concepts so that they satisfy the requirement. An example is the telecommunications domain. Experiments have shown that using fairly high-level concepts (such as "connect" and "disconnect" in the telecommunications domain) mean that introducing new requirements (such as "onhold") are difficult, because the level of abstraction is too high. Shifting the level of abstraction lower (to concepts based on signals on telecommunications lines, for example) eases this problem, at the expense of more complicated specifications.

A problem with service requests encapsulated in messages is domain terminology. Consider two services which need to communicate in order for the first service to request the second service to perform call-redirection, which consists of three main entities:

- Caller telephone number;
- First callee telephone number;
- Second callee telephone number.

Imagine that the first two telephone numbers are telephone company A numbers and the third telephone number is a telephone company B number. This means that the call-redirection service of telephone company A needs to inter-operate with the services of telephone company B. Specifically, the call redirect service of A should result in a request to the connect call service of B. There are two assumptions being made here;

- Messages are always sent to the "correct" FSE, which is able to interpret them;
- Both services are using the same DEM and domain data semantics.

Both of these are bad assumptions to make. It must instead be assumed that messages may indeed be sent to the wrong service and that services may not be talking about the same thing, or may be talking about the same thing but using different models to do so. In order for a service to decide whether an incoming request is within its own capabilities, there must be some way for the service to identify when either incompatible terminology or domain models are being used. This reasoning could be based upon the DEM. However, there must be some way to be able to uniquely model a domain. The DEM, as a concept, on its own is not enough to accomplish this.

It is assumed that, with the adoption of a common syntax for all messages (see chapter 5 section 3.1.3), parsing will always be successful. That leaves message interpretation and execution. Message interpretation is very important

because it allows a service to determine how the software entities in the message relate to the software entities which the service uses. There are a number of options:

1. The message has been sent to the correct service, and *all* of the message software entities are interpretable;
2. The message has been sent to the wrong service. For example, a message to put a user on hold that is sent to a graph layout service is clearly a fault. In this case, interpretation will fail because the service is unable to relate *any* of the software entities in the message with the software entities that it knows about;
3. The message has been sent to the correct service, but some of the software entities can't be interpreted. This will occur when the message is requesting the service to perform a task that is within the evolution space of the service. For example:
   - The message is requesting a sort service to sort data with which it is not familiar;
   - The message is requesting a sort service to sort data, given a set of non-functional requirements that are not within the constraints offered by the sort functional software entity.

This protocol is only useful for changes in messages that satisfy the following constraint:

Message evolution/extension constraint: changes in the message can only occur within the context of the functionality of the message. In other words, the evolution is constrained to evolution of parameters (including the "this" parameter for object-oriented messages).

Evolution of the parameters is in terms of *IsA* relationships. The task aspect of the message is primitive i.e. can't be refined or evolved further.

In any domain, it is assumed that there is a basic primitive set of data types. For example, in the telecommunications domain this set consists of the telephone number type. The rest of the domain is built upon these primitive building blocks. This assumption can be used as the basis of deciding whether a message is interpretable by a functional software entity, because if the message contains primitives concepts of the domain then the software entity has a chance of interpreting it. The assumption is that the functional software entity making the service request specifies everything in terms of these primitive concepts. If not, then there is no chance of this working. This argument does, however, depend on every future change being expressible in terms of the primitive concepts and that the primitive concepts are complete to start with. This is problematic because changes outside the system, such as changes to hardware elements in a telephone system, may introduce new primitive concepts. For example, the introduction of a new tone on a telephone line to allow an "onhold" service to be introduced.

The service proxy architecture of SEvEn allows software to recover from situations in which messages are sent to the wrong recipient service, either because the software has evolved or because of an error. Messages arriving at a service are either valid or invalid in the sense that the service is capable or isn't capable of performing them. There is no in-between, apart from perhaps tweaking non-functional aspects of the service request to enable to choose between a number of functionally similar alternatives. An example of this would be in the sorting domain, in which a particular sort service such as "Bubblesort" receives a request to sort based on a particular set of non-functional parameters (such as speed and memory requirements) which it isn't able to implement itself. It could then pass on the request to another sort

service, "Quicksort", which is able to implement the request. This is the idea behind a "service proxy", in which a service incapable of implementing a service request encapsulated in a particular message is able to pass the message on to another service by negotiating with a broker. The general procedure for this is as shown in Figure 25, in which S_1 asks S_2 to perform a task when the request should have gone to S_3 which S_1 doesn't know about). The broker holds, for each task in the software system, a mapping to the service that can perform that task.

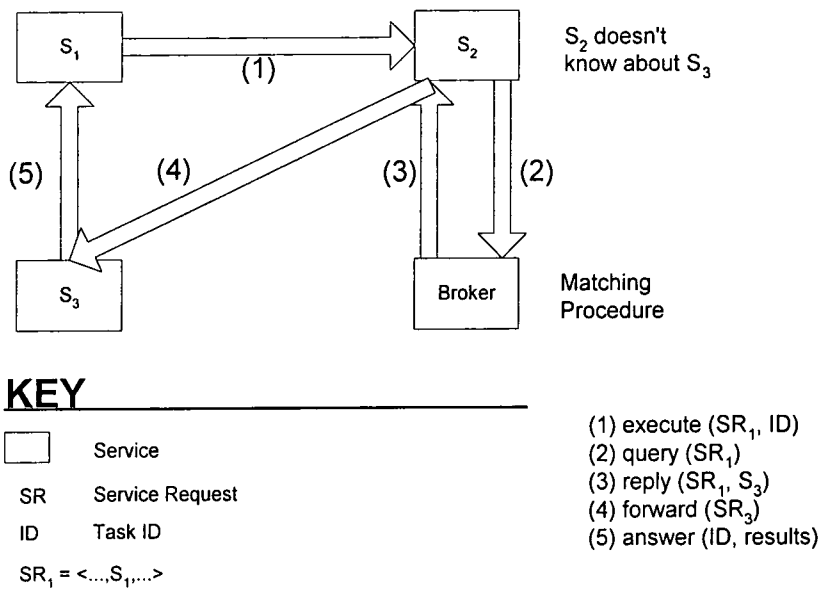


Figure 25 – A Flexibility Protocol

This broker architecture is, however, very abstract and un-constraining, making as few assumptions as possible about:

- The form of service requests i.e. whether they conform to the syntax and semantics of a particular domain language or are simply unconstrained text messages;
- The form of messages sent to and from the broker.

The broker matching procedure is important for determining the correct service to which to send a service request that has been sent by another service. Modelling is an important aspect of this and the following questions need to be asked:

- What type of model (s) should be used;
- What information should these models hold?

This choice designates the matching procedure that is used. A very simple approach is to use service names, which has a number of disadvantages:

- Service name clashes;
- Service names don't describe the service in very much detail because they're procedural;
- No matching based on semantics because they're not modelled.

The model and therefore the matching technique can be improved by including service parameters and domain information in the model and thereby in the matching procedure. However, the semantic model is still fairly basic. A further improvement would be to model aspects of the semantics of the services. There are two broad approaches to accomplishing this semantic modelling:

- Functional similarity through the use of class-based models, similar to the work of Kishimoto et al [Kishimoto95a];
- Attribute-based models.

Kishimoto et al investigate how to deal with the effects of class migration in a distributed object oriented software system [Kishimoto95a]. Their scenario involves the problem that arises when a class B using class A migrates to another node N which doesn't have class A. The aim is to find a suitable class that is functionally similar to class A in node N using an explicit class hierarchy-based model by comparing the position of classes A and B in the class hierarchy.

Their similarity measure is based on distance from the candidate class. Descendant classes are most similar, followed by ancestor classes, followed by descendants of ancestors. However, the aims of their work are subtly different from the aims of this thesis. Whereas they deal with object migration from node to node, and finding an appropriate class at the destination node, this thesis deals with changes or evolution in message service requests (or tasks to be performed) and how the software system can recover from this evolution by finding a service to perform the required task.

Their approach suffers from a need for a global class hierarchy model that all the objects in the system can access. The approach of this thesis permits the use of either a global, centrally-managed model or a de-centralised model of which different brokers maintain different parts. The disadvantage of this approach is that heterogeneity of information is more liable to creep into the model. A potential solution would be to permit, in the case of shared information, only the "owner" of the particular part of the model to maintain it. Parts of the model can then be shared without problems occurring as a result of different brokers updating the same parts of the model in different ways.

There is still a problem concerning the expression of service requests and ensuring that service requests are expressed using a shared model, or shared ontology as it is often called [Gruber93a]. This must be imposed on the tasks, so that there is a chance of them being successfully mapped to a service by the broker.

The automatic adaptation of a service in response to changes in a message is a difficult problem to overcome because new functional capabilities may be required. In addition, the mapping between the request encapsulated in a message and the services which will satisfy the request may be difficult to determine, because there is no common level of understanding in the form of shared concepts or software entities to aid the mapping. In this case, the most that can be done by the software system is to inform the user that a particular service needs to be adapted.

3.2.3.2 Adaptability With Respect to Messages Sent

This section deals with the effects on a service of evolution in messages sent by the service. The only aspect of a message which can evolve, perhaps as a result of evolution in the task to which the message is related, is evolution of the

formal parameters. Other aspects of a message, such as the location of the target task, are hidden behind an interface which shields the service from changes in these aspects of the software. Changes in, for example, the location of a service may have effects on the service due to timing issues; this is covered in section 3.2.4.

Changes in the data requirements of a service further down the call graph can have ripple effects on a service further up the call graph. Consider the example shown in Figure 26. Traditional programming languages use functional abstractions which possess a number of characteristics:

- They have formal parameters;
- They have actual parameters;
- Actual parameters are passed along with messages or procedure calls.

This architecture causes the following problem: a change in the data requirements of a function, in this case F_4, means that it acquires an extra formal parameter FP_{44}. This, in turn, means that some other function in the prefix of the call trace (functions F_1, F_2 and F_3) must provide an actual parameter to match this new formal parameter. If the function which provides this actual parameter is a long way "back" in the call graph (in this case F_1) then the formal parameter interface of each function between these two (in this case, F_2 and F_3) must be changed. These new parameters are called "pass-through" parameters and are essentially redundant.

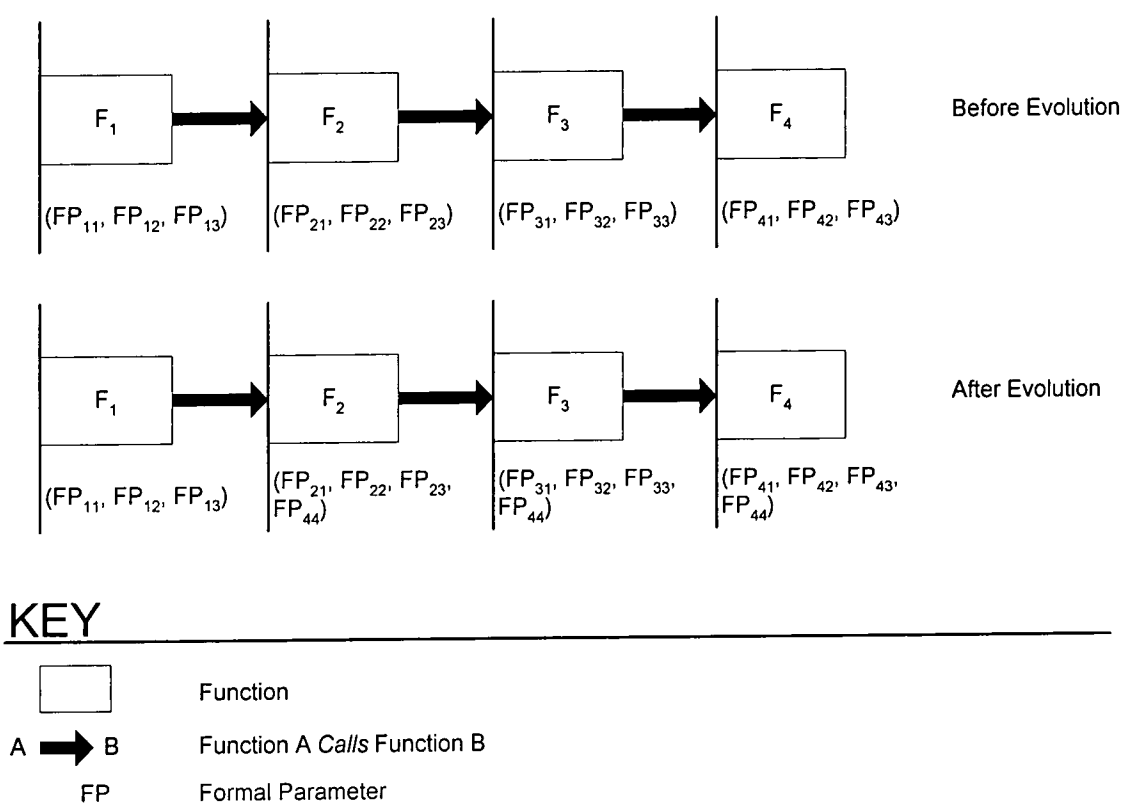


Figure 26 - Pass-Through Parameters

This problem is overcome in SEvEn by utilising a form of indirection in which an actual parameter is a "pointer" into a state space. This is advantageous because changes in the data requirements of a service will only affect those services

which need to be affected i.e. those which need to produce the required data, not those in between. This is depicted in chapter 5 figure12 and Figure 30 (with some simplification of the software entity model for clarity). Rather than having many instances of effectively the same actual parameter (as is the case in Figure 26), this approach utilises one instance coupled with direct relationships between the actual parameter and the service instances which either produce, use, change or remove it. In this way, pass-through parameters are not required and the potential ripple effects of:

- Data removal, and;
- New data required

are alleviated.

The main difficulty for services adapting to changing messages is when additional actual parameters require additional functionality. This will typically require the software engineer to write more code. The advantage of SEvEn is the ability of service instances to adapt to changes in actual parameters; a service instance may not be able to act on the evolved actual parameters but it will be able to parse them without causing errors.

3.2.4 Adaptability With Respect to Software Architecture

Most software today is written with an implicit software architecture built in. This means that issues such as:

- Patterns of communication;
- Component types;
- Connector types;
- Component visibility (and other constraints on which components can be accessed from each component), and;
- Component location;

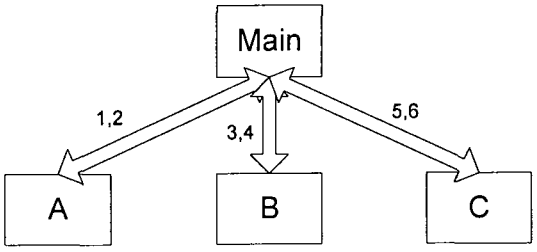
are typically implicit and hard-coded into the software, which makes it difficult to change these aspects in a well-defined way without causing ripple effects on other aspects of the code. Some researchers, notably Lieberherr et al at Northeastern University, recognise the importance of separating out the architectural aspects of software from other aspects of the software:

> "Adaptive software is generic software that needs to be instantiated by architectures." [Lieberherr96b p79]

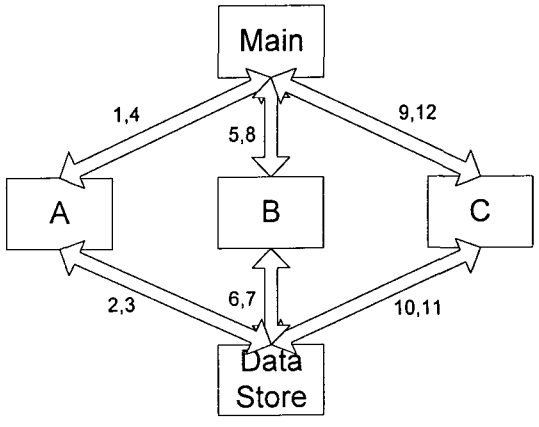
> "Adaptive software is generic software that defines a family of programs with a large variability in architecture [the interfaces and connections between interfaces]" [Lieberherr96b p79]

This section considers the adaptability of services with respect to changes in the software architecture of software systems. Software architecture deals with the topology or structure of software, including the components and

connectors that are part of the software system [Garlan93a]. Problem domains may or may not have standard architectures. For example, the user interface domain is generally modelled using an event-based architecture. Compiling domains are generally modelled using a filter-architecture, although recently shared-memory architectures have opened up possibilities for faster compiling and better optimisation by allowing each stage to work on a centrally-managed representation of the code. In contrast, the relational database domain can be modelled using a number of different architectures, including object-based and call-and-return styles. Hence, it is clear that the coupling between the problem domain and the architecture ranges from fairly high (in the case of the user interface domain) to fairly low (in the case of the relational database domain).



(a) - Filter Architecture



(b) Shared Memory Architecture

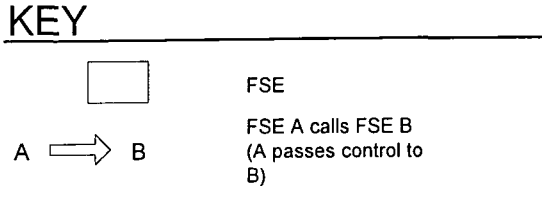


Figure 27 – A Comparison of Software Architectures

Consider what changes to the architecture actually mean:

- Changes in the components in the architecture;
- Changes in the connectors in the architecture.

Such changes can be fairly wide-ranging in their impact on other aspects of the software system, so that changing the architecture in order to improve compatibility with another software system, or the speed of the application can be fairly

devastating, even though architectural changes may be quite infrequent. Is there some way of de-coupling the core code from the architecture, in order to increase the adaptability of the core code to changes in the architecture?

Call Number	Description
1	GetData ()
2	ReturnData
3	SortData ()
4	ReturnSortedData
5	OutputData
6	Acknowledgement

Table 6 - Filter Architecture Service Calls

Call Number	Description
1	GetData ()
2	PutData ()
3	Acknowledgement
4	Acknowledgement
5	SortData
6	SortData
7	Acknowledgement
8	Acknowledgement
9	OutputData
10	OutputData
11	Acknowledgement
12	Acknowledgement

Table 7 - Shared Memory Architecture Service Calls

Why does changing the software architecture affect the functional aspects (the FSEs) of the code? The relationship between Garlan and Shaw's component and connector architecture entities and the software entities used in this thesis is shown in chapter 5 figure 6. Ideally, the FSEs should be de-coupled from any architectural information and dynamic binding used in order to link them either at compile-time or run-time. In this way, like the other approaches described in this thesis, software architecture information is hidden behind an interface so that most changes in software architecture don't affect the FSEs in the software system. The exceptions to this are detailed below.

As an example problem domain and architecture, consider Figure 27, Table 6 and Table 7. The numbers show the sequence of calls made between FSEs. Figure 27 (a) relates to Table 6 and Figure 27 (b) relates to Table 7. The transformation between the two architectures can be expressed in terms of sequences of message mappings, in which the sequence of messages constituting the filter architecture can be mapped to the sequence of messages constituting the shared memory architecture. In effect, whole conversations (to use Bradshaw's terminology [Bradshaw96a]) are being

transformed. These conversation mappings can be represented using a DEM mapping, in which data entities are replaced by messages.

In order to de-couple the FSEs which constitute the core code (in this case the input, sort and output services) from the architecture and provide a late binding between the two:

- Look at the similarities between the two implementations;
- Build an interface based on these similarities and hide the differences (or changes) behind this interface, so that the differences don't affect the core code. These differences then form part of the interface and evolution space of this interface, which is circumscribed by a set of parameters.

In the case of software architecture, differences are based on differences in the components and connectors as described above. However, to start with, we consider only changes in the connectors (since the components stay the same), which are of the following types:

- Sender of message;
- Recipient of message;
- Control tag of message;
- Data tag of message.

The similarities can be extracted out to form an interface between the FSEs and the software architecture information, based on providing an interface which is a generalisation of a number of software architectures as shown in Figure 28.

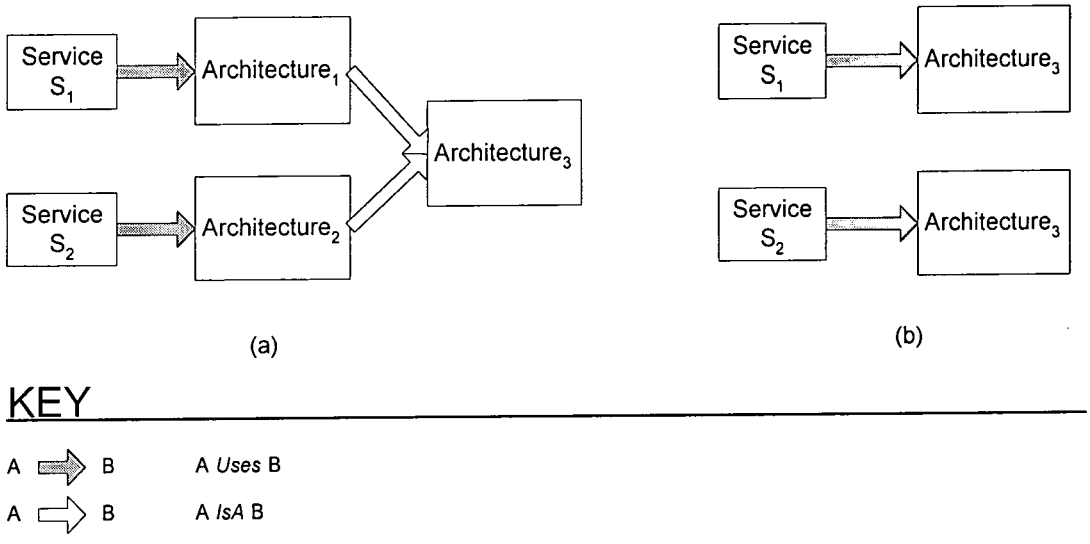


Figure 28 - De-coupling Software Architecture

Of course, this will only work if such a generic interface as "Architecture$_3$" can be constructed. The main requirement for this is that the architectures must be compatible, as in the example above. However, there are many occasions where this is not the case. For example, changing from a blackboard architecture to a filter architecture is impossible unless the requirements change drastically, because the two architectures are essentially incompatible. Up until now, adaptability

with respect to software architecture has been confined to adaptability with respect to the connector aspects of software architecture. However, software architecture also consists of component types, such as schedulers in blackboard architectures, event controllers in event-based architectures and filters in filter architectures. These aspects are more difficult to change, because they often form part of the requirements and application domain of a software system.

This leads on to another aspect of adaptability with respect to software architecture which is concerned with extensibility and the task-service separation of concerns. This form of extensibility allows a new service to be slotted into a running software system, providing:

- It *Implements* an existing task abstraction, and;
- The existing task abstraction is being used by some existing process in the software system.

As an example, consider the re-configuration of a software system so that a routing service is replaced at run-time. It is assumed that an appropriate routing task software entity has been created and integrated into an existing process. The integration of a new routing service then requires that the *Implements* relationship with the routing task be fulfilled, which means that the task interface is satisfied.
However, any service which introduces radical new behaviour i.e. it doesn't implement an existing task software entity, will be difficult to integrate.

Services also assume the following software architecture characteristics of other services which they use:

- The locality, and;
- Names.

Hence, a sort program's main service assumes that a service called "BubbleSort", accessed through a local service call, will sort its data. These assumptions will break if the service changes its name or the software evolves into a distributed architecture in which the "BubbleSort" service resides on a remote machine. Messages remove assumptions about the locality of services and the task-service dichotomy provides a way of separating information about what to do from how to do it.

This section has shown how some aspects of the software architecture of a software system can evolve without affecting the services in the software system. The approach is based on hiding software architecture information from the services behind an interface which localises changes in the software architecture. The "software architecture interface" consists of:

1. The task or service to call;
2. Duration constraints. For example, requests made of the interface must be completed within a certain time period.

However, there are situations in which changes in the software architecture can affect this interface and cause unavoidable ripple-effects on services. For example, 2 can be affected by changes in the patterns of communication. These ripple effects are unavoidable because they break the requirements made of the services and must inevitably result in either:

- A relaxation of the requirements;
- Further evolution of the software architecture in order to re-satisfy the requirements, if possible.

Changes in the architecture affect the patterns of communication and may result in the following change types:

- Less messages being sent;
- More messages being sent;
- Changes in the location of FSEs.

The limitations of this form of adaptability are imposed by the need for the architecture to not break any requirements imposed on it. For example, a sort program is typically implemented in terms of a filter architecture consisting of three services (input, sort and output services) and a process. Changes to the architecture can't break the architecture by changing its filter characteristics, for example, because this would break the requirements of the sort program as a whole. Hence, the limitations imposed on this form of adaptability are as follows:

- Component types can only be specialised and new instances created of them;
- In general, new top-level component types are not allowed because they will break the architecture, unless mixed, or hybrid, architectures are permitted. For example, adding a scheduler component type to an event-based architecture transforms the architecture into an impure event-based architecture, a hybrid of an event-based architecture and another (unknown) architecture;
- Any changes in the communication patterns must conform to both behavioural requirements and non-functional requirements such as speed of execution and, possibly, security (if there is a requirement that certain data is not shared with particular services).

In conclusion, adaptability of services with respect to software architecture has been improved by removing assumptions regarding:

- Patterns of communication;
- Location of services.

3.3 Service Instance Evolveability

3.3.1 Adaptability With Respect to Actual Parameters

A process state space provides formal parameters to a service. The decision to use a centralised data access model as a parameter-passing mechanism is based on the following observation:

> Any extra parameters that are required don't need to result in changes to the client service's code because the server service simply examines the state space for an appropriate formal parameter.

A service is dependant on the DEMs which are its formal parameters. These DEMs are prone to evolution as discussed in chapter 6 section 4.1. This evolution can have an effect on the service, akin to changes in formal parameters in traditional programming languages such as C. For example, consider the BubbleSort function written in C-like pseudo code in Figure 29.

```
function BubbleSort (int[] Data) {
    for (int j = sizeof (Data) - 1; j > 0; j--) {
        for (int i = 0; i < j; i++) {
            if (Data[i] > Data[i+1]) {
                swap (Data, i, i+1);
            }
        }
    }
}
```

Figure 29 - A BubbleSort Function

Notice that BubbleSort makes a number of assumptions about the formal parameter "Data":

- Data is an array of integers;
- The comparison operator is integer comparison.

If the data to be sorted is an array of integers with different ordering semantics (for example, a code of some kind which has different ordering than integers) or consists of multiple keys, then BubbleSort will have to be adapted. Eliminating these assumptions is possible (by adopting the use of a comparison operator which is passed as a parameter to BubbleSort, for example, and by utilising a more generic formal parameter which doesn't impose constraints on the form of the data to be sorted) and would improve the adaptability of BubbleSort with respect to changes in its formal parameters.

The use of a graph-based model for representing DIMs and DEMs provides a generic model and helps to overcome the problems of dependence on a particular type, because the graph-based model is flexible enough to represent many

different types of data. Obtaining complete adaptability of services with respect to formal parameters is, however, difficult because formal parameters may result in extra functional capabilities being required. This is typically the case when the service has been incompletely modelled with respect to its data. For example, the developer of a sort service may have failed to build in functionality for sorting particular types of data. Another cause is when the data itself does not adequately model what is required, and so the data model must be changed. This will affect the service, which must be changed to be able to use the new data model.

However, there are two improvements that SEvEn utilises which improve the adaptability of services with respect to formal parameter changes:

- The use of a generic data model such as DIMs and DEMs allows data to change (often quite drastically) without affecting particular aspects of the service. Specifically, the evolved data can still be parsed because of the genericity of and lack of assumptions made by DIMs and DEMs. This means that new types of data can be passed to a service, which will be able to parse and interpret them. However, the service may not possess the functionality to be able to use this data in the way it is intended, but it will be able to determine its lack of capabilities rather than simply breaking because of the change;
- The model is extensible and allows new software entities and relationships to be added (along with appropriate semantic machinery) in order to improve the level of information about data entities.

A change to part of an instance model may affect existing services whos parameters have access to this changed instance sub-model. However, if the mapping between the old instance model and the new instance model is known (using DEM mappings), then such mappings can be applied to the hard-coded assumptions about the structure of the data in the services themselves. This provides an approach to overcoming ripple effects whereby changes in the structure of data affect the services.

The real problem is when formal parameters change such that a change is required in the service, which can be of two types:

- Behaviour-affecting;
- Non-behaviour-affecting. For example, a change in the formal parameters to a graph layout service results in a change from a centralised algorithm to a distributed algorithm.

The explicit modelling of the dependencies of service instances on actual parameters (represented by DIMs in the state space of the process) allows the elimination of so-called "pass-through" parameters, as Figure 30 shows.

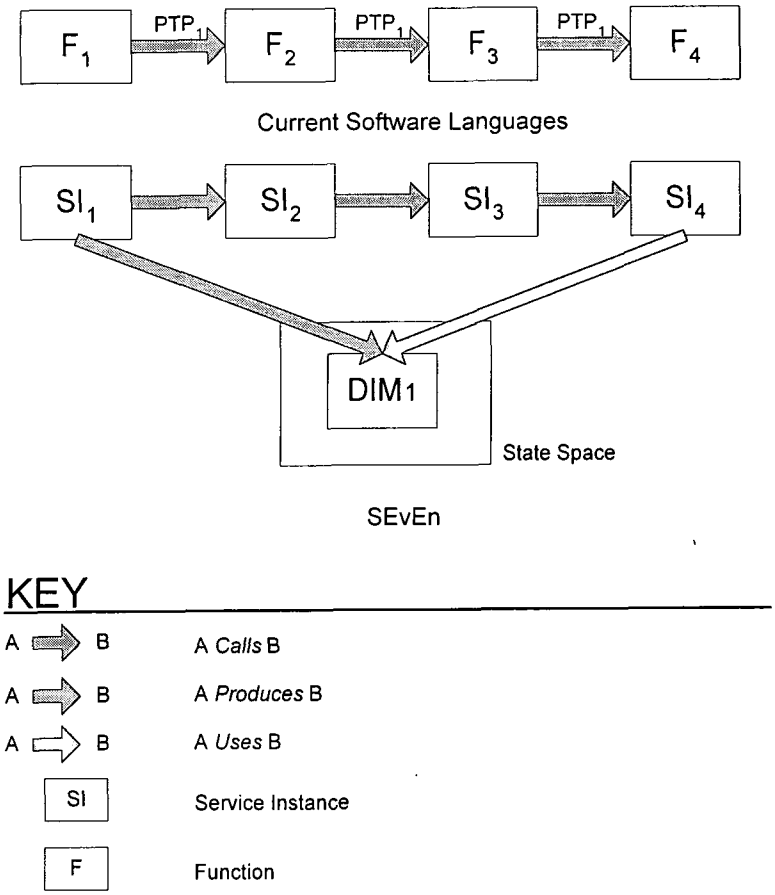


Figure 30 - The Elimination of Pass-Through Parameters

Data use in services can be viewed as DIM paths or as data access services. Consider chapter 6 Figure 4, in which data usage using DIM paths is shown in boldface.

3.3.2 Adaptability With Respect to Service Instance Evolution

Service instances depend indirectly on other (server) service instances, a relationship which depends on the relative positions of these service instances in the call graph. Two service instances are related in this way if the former is *after* the latter in the call graph. The *after* relationship is defined between two service instances, SI_1 and SI_2 such that SI_1 is the parent and SI_2 is the child, as follows: SI_2 lies on a direct path in the call graph from SI_1. The evolution of SI_1 can potentially affect SI_2 in the following ways:

- Changes in SI_1 which involve it changing DIMs that SI_2 uses;
- Changes in SI_1 which involve it using resources that SI_2 also uses, as shown in Figure 31 (using a simplified call graph model which omits task and message instances).

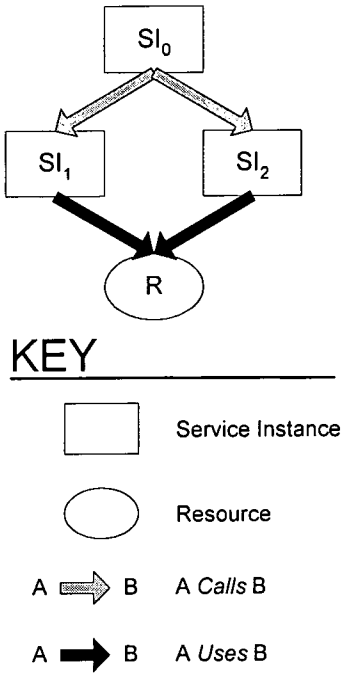


Figure 31 – Service Instance Conflicts

Before	After
Produces	Null i.e. remove.
Removes	*Uses, Updates*
Uses	*Removes, Updates*
Updates	*Removes, Uses*
Null	*Produces, Uses, Updates*

Table 8 - Evolution of *Produces*, *Removes*, *Uses* and *Updates* Relationships

Table 8 shows how *Produces*, *Removes*, *Uses* and *Updates* relationships can evolve. "Null" indicates "no relationship". So, for example, a *Produces* relationship can't evolve into any of the other three (because they require the resource to exist, which is the purpose of the *Produces* relationship). The last entry in the table indicates that any relationship can be brought into existence. In effect, the table depicts a state machine model of how these relationships can evolve.

Table 9 shows the possible relationships that can exist between two service instances, SI_1 and SI_2, with respect to resources. X's indicate that this type of relationship is not possible; for example, cell 1 shows that SI_1 and SI_2 can't both Produce the same resource. The numbers are references used to discuss the effects of resource usage in SI_1 on SI_2, as shown in Table 10. The "Effects" column shows how evolution in SI_1's resource usage affects SI_2 by referencing the numbers in Table 9.

		SI_2			
		Produces	*Removes*	*Uses*	*Updates*
SI_1	*Produces*	X	1	2	3
	Removes	4	X	X	X
	Uses	X	5	6	7
	Updates	X	8	9	10

Table 9 - Resource Usage Relationships Between Service Instances

	SI_1 Before	SI_1 After	Effects on SI_2
A	*Produces*	Null	1, 2, 3
B	*Removes*	*Uses*	4
C	*Removes*	*Updates*	4
D	*Uses*	*Removes*	5, 6, 7
E	*Uses*	*Updates*	6, 7
F	*Updates*	*Removes*	8, 9, 10
G	*Updates*	*Uses*	9, 10
H	Null	*Produces*	None
I	Null	*Uses*	9, 10
J	Null	*Updates*	9, 10

Table 10 - Effects of Resource Usage Evolution in SI_1 on SI_2

As can be seen from Table 10:

- Any change in SI_1's usage of a *Produces* relationship (A) will affect SI_2;
- Any change in SI_1's usage of a *Removes* relationship (B and C) will affect SI_2;
- Most changes in SI_1's usage of a *Uses* relationship (D and E) will affect SI_2. The exception to this is E, which won't affect SI_2 if SI_2 removes the resource;
- Most changes in SI_1's usage of an *Updates* relationship (F and G) will affect SI_2. The exception to this is G, which won't affect SI_2 if SI_2 removes the resource;
- H results in the creation of the resource which SI_2 is using and hence should not affect SI_2;
- I and J assume that the resource has been produced further back in the call graph. They can affect SI_2, apart from the case in which SI_2 is removing the resource (8).

3.4 Task Evolveability

3.4.1 Adaptability With Respect to Service Evolution

A task guarantees a particular requirement to any service clients. This places a constraint on any services which *Implement* the task. The requirements which the task encapsulates may not, however, be completely specified. There are two aspects to this:

- The contract between two services which expresses the clients' (or callers') requirements of the server (or callee) may intentionally leave aspects of the requirements open. For example, speed and duration, parts of the callee service which only get used for a particular combination of parameters which contradicts the caller's explicit requirements i.e. the callee isn't fit for its purpose;
- Requirements generally assume that anything which isn't specified is not permitted. For example, a client service requires a graph layout service to lay out a 2D graph. The requirements specification states this requirement but assumes, for example, that the graph layout service will not, in addition to laying out the 2D graph, change the colour of the nodes in the graph.

These characteristics of requirements determine the effects of evolution in a service on the task interface and client services of the task. For example, client requirements are often very constraining and therefore leave very little room for any evolution in server services which the client calls, which don't break the service interface or contract between the client and server. This means that evolution in the server service which isn't simply a re-implementation may break the contract and result in the client having to evolve, by using another service.

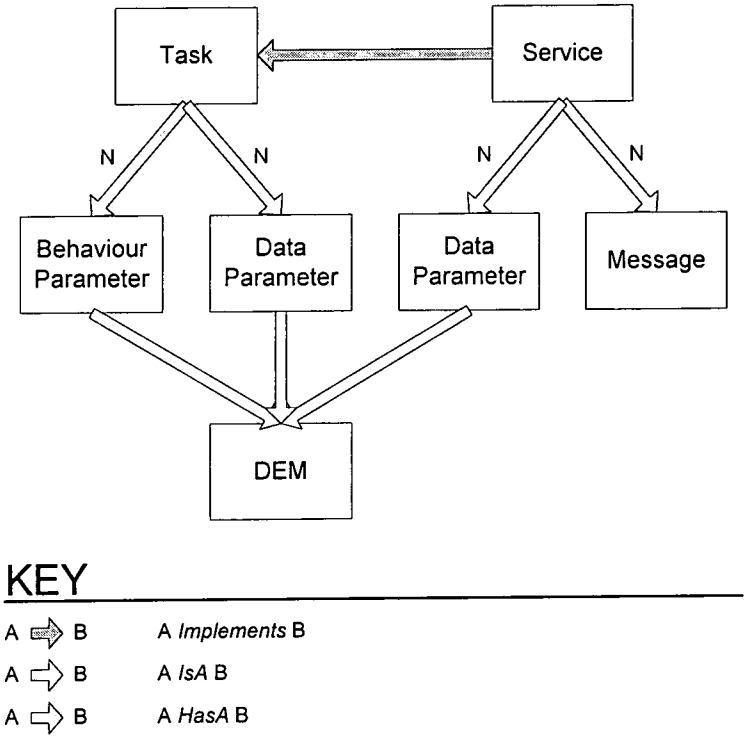


Figure 32 - Task and Service Relationship

Figure 32 shows the relationship between a task and a service. Section 2.1 described the evolution space of a service. Any evolution of a services' data formal parameters will have a ripple effect on the task to which the service is related. However, the interaction between message evolution in the service and the task is somewhat more complicated by the observations made above about requirements. A task assumes a particular behaviour of the service to which it is related. Changes in service behaviour (where behaviour is taken to mean the input-output relation of the service, or in terms of changes in the configuration of the software system's environment) may have an effect on the task. Changes to service behaviour can be classified as follows:

- Same behaviour;
- Behaviour extension, whereby the original behaviour is preserved but the service does more. The behaviour extension doesn't conflict with the existing behaviour;
- Behaviour removal, whereby some part of the behaviour of the service is removed;
- Behaviour change, whereby some part of the behaviour of the service is changed, so that taken as a whole the new service behaviour conflicts with the old service behaviour.

Behaviour extension of a service can be modelled by the addition of more messages to a service, such that the new messages don't produce conflicts with existing messages (for a definition of message conflicts, see chapter 5 section 3.1.3.1). Behaviour removal can be modelled by the removal of existing messages in the service. Behaviour change can be modelled by adapting *existing* messages in the service. All of these changes will change the behaviour of the service in some way. However, it is difficult for software to determine if a service *Implements* a particular task and, by implication, it is also difficult for software to determine the effects of changes in the messages of a service (resulting in a change in behaviour of the service) on the *Implements* relationship. This could be approached by comparing the input-output relation of the service to the requirements but would have to check every permutation of the input, which may be infeasible. Hence, this requires the intervention of a software engineer.

So, in summary, whether or not a service change affects any tasks to which it is related is dependent on the definition of the task itself. A more abstract task definition, or a task definition which has few constraints on it may allow more room for service behaviour to change without the change affecting the task to service mapping. For example, task A may allow dependent services to extend their behaviour without affecting the task definition. For this to work, the task definition would have to be annotated with information that indicates the limits of what kind of behaviour it will allow.

3.5 Message Evolveability

3.5.1 Adaptability With Respect to Service Evolution

Since behavioural "contracts" exist between two services, messages are fairly static software entities which only encapsulate:

- Architectural requirements (such as whether synchronous, asynchronous, local or remote semantics are required, and where the broker resides);
- The required task;
- Parameters.

The invalidation of architectural aspects of the code is outside the scope of this thesis, which assumes new requirements won't drastically alter a software system's architecture. Hence, changes from a blackboard architecture to a filter architecture are assumed not to occur. The reasoning for this is that such a change in architecture is likely to be a result of major changes in requirements or a change in the application domain itself.

The "required task" is the task that the client service is requesting the server service to perform. This will be affected by changes in the task requirements of the client service and can't be avoided. Adaptability to this kind of change is not possible and so will result in ripple effects.

Changes in parameters are covered in the section on service evolveability.

4 Summary, Discussion and Conclusion

This chapter has discussed the functional aspects of the SEvEn model, which consists of:

- Task and task instance software entities;
- Service and service instance software entities;
- Message and message instance software entities.

along with their evolution spaces and evolveability. A point to note is the limited adaptability of instances (task, service and message instances) with respect to evolution in the corresponding software entity (task, service and message), a conclusion that was drawn in chapter 6 when DIM adaptability with respect to DEM evolution was discussed. The reason for this is the high coupling between instances and entities, a coupling that limits the adaptability of instances with respect to entities.

This chapter has explored FSE evolution and evolveability, applying the evolution space concept to tasks, services, messages and their instances in order to arrive at a taxonomy of change types for each one. Where possible, the adaptability of the software entities has been improved through the limiting of assumptions and, where complete adaptability is not possible the effect of the identified change types on dependant functional software entities and instances was then explored by relating the change types to their effects on the interface of the entity or instance. In addition, the adaptability of services with respect to changes in particular characteristics of software architecture was explored. Finally, FSE flexibility (the ease with which the FSE, or functional, sub-model of a software system can be changed with respect to new requirements) based on a broker architecture has been discussed, with particular emphasis on the nature of the "glue" connecting services and the "visibility" of service calls.

It is interesting to comment on the notion of dynamic insertion of components into a running software system. Blackboards support this within a limited domain, that of data-driven components called knowledge sources. The event-driven model used by blackboards could be extended to allow components to be executed (or scheduled) based on a wider environment. However, highly context-dependant components will work only in particular contexts, obviating the need for them to be implicitly called when a particular event occurs. In addition, well-defined algorithms are best suited to a traditional means of modelling in which control-flow is explicit. For example, expressing a sort program using an event-driven architecture is convoluted because the algorithm for doing it is well-defined. Nothing is gained from the event-based approach in this case. However, there may be domains in which such an approach bears fruit, as blackboard architectures attest to. However, the fact that event-based architectures are not well-suited to expressing all types of application means that their benefits with respect to the dynamic insertion of components can't be utilised. Even in an

application for which an event-based model is suitable, the dynamic insertion of components is dependant on the new component being expressible in terms of existing components in the software system for the dynamic insertion to be straightforward (for example, the insertion of a sort component into a sort architecture is easier than the insertion of a graph layout component into the same architecture). A special case of this is when the dynamically-inserted component is a specialisation of an *existing* abstraction in the software system, in which case the interface for insertion already exists and insertion is simply a matter of ensuring that the interface of the new component and the insertion interface are linked. The nature of this link would be dependant on the domain, so that insertion of a sort component would have to ensure that the data in the insertion interface is linked to the data parameter of the sort component.

Dynamic insertion is partially supported by the task and service abstractions, for which the task abstraction provides the hook or interface for insertion and the service abstraction provides the actual component for insertion. As an example, consider an architecture for a sort program consisting of four *abstract* components; main, input, sort and output. Main calls input, sort and output in turn, passing the data to be sorted between them as necessary. Dynamic insertion of components in this example could be the dynamic insertion of *concrete* input, sort and output components, such as a "Bubblesort" service.

The task-service modularisation provides for extensibility characterised by the following:

1. Existing interfaces aren't changed;
2. New interfaces aren't created;
3. Existing interfaces aren't removed;

Some capability which allows interfaces to be dynamically created and related to services would be required. This, in turn, would require the existence of semantic information or "integration knowledge" that describes (perhaps in an abstract manner) how to link new software entities to existing software entities. The form of this knowledge will be dependant on the type of software entity. This will be a simpler task *within* a domain because the domain provides the required terminology for the integration knowledge. In multi-domain applications, however, this will be more difficult as the notion of semantic heterogeneity creeps in.

Chapter 8

Evaluation

1 Introduction

Chapters 4, 5, 6 and 7 described an architectural framework, SEvEn, with improved evolveability over existing software architectures, models and languages (hereafter collectively termed "software models"). This chapter aims to evaluate SEvEn against the set of criteria set out in chapter 1 and against existing software models. In particular, the chapter aims to show how the software entities used in SEvEn improves the evolveability of software and locality of evolution.

2 An Analysis of the Evolveability of Existing Software Languages, Models and Architectures

This section analyses a set of existing software models for their evolveability, in order to contrast them against SEvEn. This is performed by comparing each software model in turn against the evaluation criteria in chapter 1. For each software model considered, the following is performed:

- The identification of a set of software entities (the abstractions) which compose the software model (in Garlan and Shaw's terminology, the so-called components and connectors [Garlan93a, Shaw96a]);
- The dependencies between the software entities;
- An analysis of the flexibility of the software model;
- An analysis of the adaptability of the software model;
- An analysis of the extensibility of the software model;
- An analysis of support for localisation of evolution.

As pointed out in chapter 1, measures of evolveability (adaptability, flexibility and extensibility) are based on the identification of particular types of evolveability, and not on any quantitative measure of evolveability.

Section 3 then goes on to analyse the proposed architectural refinements discussed in chapters 4, 5, 6 and 7 to analyse SEvEn against the same set of criteria in order to determine how SEvEn improves over existing software models with respect to evolveability. This proceeds on a case by case basis using examples to show specific improvements.

2.1 Functional Models

The software entities comprising a functional model of software (such as C or Pascal) are shown in Table 1, and their dependencies are shown in Figure 1. As can be seen, the abstractions consist of four main types allowing the direct

expression of functional and data abstraction but not process abstraction (except the "main" function in C, which means that process abstractions are represented as function abstractions), although thread libraries allow the expression of process aspects in an indirect manner.

Software Entity	Software Architecture Entity Type
Function	Functional Component
Primitive Data Type	Data Component
Record	Data Component
Function Call (local procedure call semantics)	Connector

Table 1 - Functional Software Entities

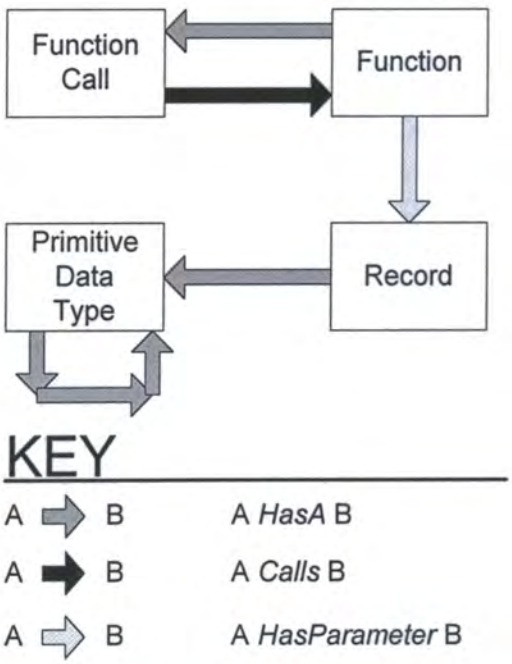


Figure 1 - Functional Software Model

Functional models typically build in a lot of assumptions and design decisions:

- Function calls are assumed to be local and synchronous;
- Tight coupling of functions to software architecture;
- Tight coupling of functions to parameters;
- Tight coupling of functions to data structure;
- Tight coupling of functions to functions, so that information of the form "function F_1 can provide functional capability C_1" is implicit and hard-coded.

However, functional models are fairly extensible (they don't impose many constraints on how software can be extended though they make it difficult to integrate extension changes into an existing software system) and flexible (constraints

which prevent new functions and data structures to be added to a software system are limited). Their flexibility is let down by the need to re-compile, in most cases, although interpreted functional languages don't suffer in this regard.

Functional software models don't provide any means to determine the effects of particular kinds of change in an abstraction on other abstractions. In general, ripple effects have to be dealt with in an ad-hoc manner during the source code compilation process, although tools exist which allow potential ripple effects to be pre-determined (see [Turver93a] for a good overview of this research area).

2.2 Object-Oriented Models

The software entities comprising an object-oriented model of software (such as C++ [Stroustrup94a] or Java [Flanagan97a]) are shown in Table 2 and Figure 2.

Software Entity	Software Architecture Entity Type
Class	Data Component
Object	Data Component
Class Method	Functional Component
Class Data Member	Data Component
Object Data Member	Data Component
Message (local procedure call semantics)	Connector

Table 2 - Object-Oriented Software Entities

Proponents of object-oriented models often promote their improved evolveability, without much evidence to support this viewpoint. However, object-oriented models are an example of a software model family which, whilst geared towards improving the ease of software development by the use of encapsulation, inheritance and polymorphism, sacrifice ease of evolution. The reasons for this are discussed in this section, which analyses the evolveability of object-oriented software models and entities in order to determine the effectiveness of object-oriented software with respect to software evolution.

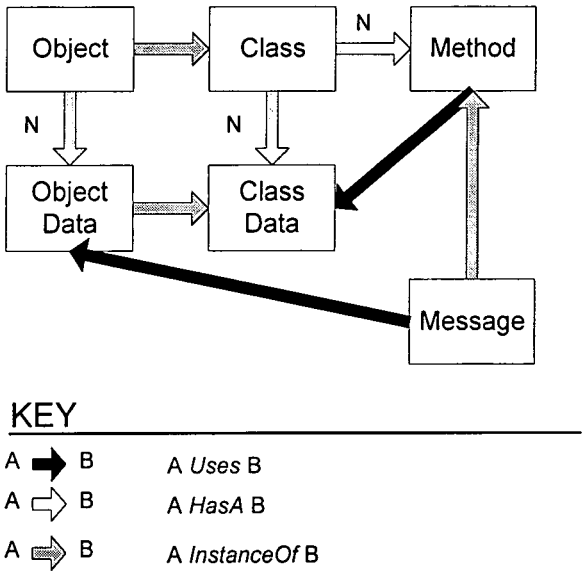


Figure 2 - Object-Oriented Software Entity Model

Object-oriented models are inflexible for a number of reasons:

- A class model is difficult to change because of the constraints imposed on it by encapsulation and inheritance;
- Method visibility is a problem.

Encapsulation provided by classes hinders changes to the class model by restricting access to class members. Consider the class model shown in Figure 3. Method y changes and now requires the use of a method with class C as a formal parameter (this method could either be a method in C, or a method in which C is a parameter) but class B has no reference to class C. Where should the reference come from:

- Change class B to include a reference to class C;
- Pass class A's reference to C as a parameter to y.

Hence, object-oriented designs are inherently difficult to change.

The constraints imposed by the class model limits method calls to those that:

1. Can be accessed directly from the class;
2. Can be accessed from a referenced class;
3. Can be accessed from a superclass of any of the classes identified in 1 and 2.

So, the method visibility of method x consists of the set of methods which satisfy, for each parameter P_i of x:

- M_{aj} where a $\in$ superclasses (classof (P_i));
- M_{bj} where b $\in$ classof (references (classof (P_i)));
- M_{cj} where c $\in$ superclasses (references (classof (P_i))).

Hence, flexibility with respect to method calls is severely impaired. Each set of requirements will map to a number of different object-oriented models, each with it a particular level of flexibility with respect to a particular set of requirements. The trick is in choosing the object-oriented model that both satisfies the current requirements and allows future requirements to be met with as little re-modelling as possible. Of course, there are no heuristics or techniques for doing this because future requirements can't, in general, be determined, and so the object-oriented model which is chosen is often dependent on satisfying existing requirements only, with the choice between different potential models being based on other factors such as efficiency, security, or the need to integrate with other systems.

The adaptability of object-oriented models is interesting because of the effects of (method and data) encapsulation and inheritance which this model introduce as abstractions, and the problems they pose for changes which involve changing *existing* parts of the model. For example, the fragile base class problem and related problems deal with the fact that classes are not very adaptable to changes in their superclasses which result in non-additive operations such as removal of class elements or adaptation of class elements. Some have attempted to partially overcome such shortcomings by employing "design for change" approaches which impose rules on the design process in order to ease evolution of the model. One approach utilises a technique called "managed inheritance" [Mikhajlov97a], which involves imposing rules on inheritance that reduces the effects of the fragile base class problem. However, there will always be changes which will affect dependants of the class (those classes that reference or inherit from the class), such as changes which result in the removal of class elements on which descendants of the class depend.

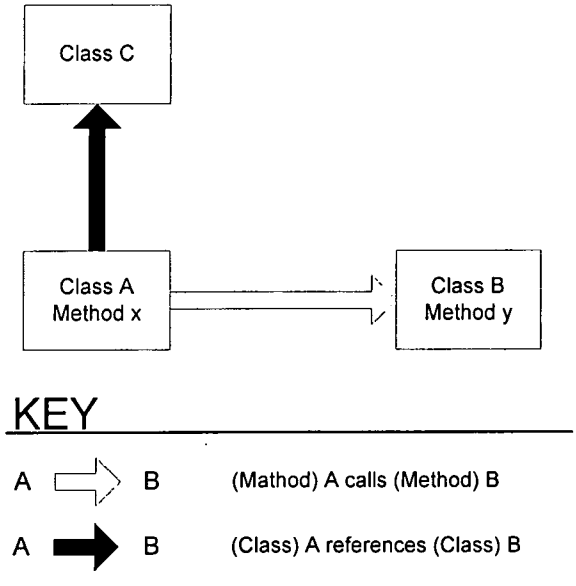


Figure 3 - Encapsulation and Method Calls

Object-oriented models inherently provide support for extensibility because the class abstraction coupled with inheritance provide a hook for adding new functionality and data. The constraint on this is that the new functionality and data must, in general, be a semantic sub-type of the super-class, although this is only partially enforced by object-oriented models through type-checking. This is probably not a problem because the reason for using this form of extensibility in the first place is precisely in order to perform this kind of extension change.

The hard-coding of knowledge about which functional abstraction can perform a particular task causes problems when the assumption is broken. Object-oriented models complicate this somewhat because the targets of messages are classes (i.e. messages are sent to classes), which means that task information is tied to both class and method. This makes it more difficult to change this type of dependency.

Object-oriented models decrease adaptability for methods with respect to methods which they call. For example, moving method, M_1, from class, C_1, to class, C_2, will have an effect on any other methods using M_1. The classes of which these methods are a member will then have to be changed in order to provide a reference to C_2 so that M_1 can still be called. Hence, changes in the location of a method results in class interface changes which can have far-reaching effects on the class structure. Existing object-oriented models provide little support, in terms of adaptability, for this.

2.3 Blackboard Architectures

Functional and object-oriented software models are fairly generic in that they can be used to model a wide range of requirements. A blackboard architecture, on the other hand, is what this thesis has previously described a niche model (chapter 6) because it was designed for a specific purpose, namely data- or environment-driven control [Corkhill91a].

As figure 6 in chapter 3 shows, a blackboard model[1] consists of the software entities shown in Table 3. The scheduler is the main control element which drives the blackboard by activating knowledge sources (which are a type of functional component) whose trigger- and pre-conditions are met. These conditions are typically functional components which return a true or false value depending on the state of the blackboard, which consists of objects, classes and relations. Events, which are generated by knowledge sources and correspond to particular types of changes to the blackboard, typically form the trigger conditions of knowledge sources. The main connector type is simple synchronous, local message-passing, depending on the underlying model used; whether it be a functional model or an object-oriented model.

Blackboard architectures are extensible because they permit the dynamic addition of a new knowledge source to a blackboard, which will then be automatically used whenever its trigger- and pre-condition are true. However, this extensibility is limited to the addition of new knowledge sources which are expressible in terms of *existing* concepts (classes) used on the blackboard. Changes which require the integration of new concepts into the existing model may hinder the extensibility of the model, depending on how easy it is to integrate the new concepts. Another related problem is the potential heterogeneity of these new concepts; a high "semantic distance" between these new concepts and the existing model will make integration difficult. In addition, evolution which requires changes to the architecture will be as difficult to make as with any other architecture, in general (ignoring the ease or difficulty of making changes for a particular set of requirements and design).

[1] Rather than "*the* blackboard model" because there exist many different instances of the blackboard model. In Garlan and Shaw's terminology, blackboard models form an "architectural style".

Software Entity	Software Architecture Entity Type
Knowledge Source	Functional Component
Trigger Condition	Functional Component
Pre-condition	Functional Component
Obviation Condition	Functional Component
Class	Data Component
Relation	Data Component
Event	Data Component
Scheduler	Functional Component
Object	Data Component
Message	Connector

Table 3 - Blackboard Software Entities

In terms of flexibility, new KSs can be added in and removed without affecting the *running* software system.

Adaptability is not relevant, since changes to a blackboard will generally be extension changes which don't break any abstraction interfaces.

2.4 Event-Based Architectures

Event-based architectures can be expressed in terms of an object-oriented or functional model or in terms of a specialised language, and consist of the software entities shown in Table 4. There are a number of ways of modelling an event-based architecture. Perhaps the most common is one in which functions post events, which are then read and the appropriate event handler dispatched to handle the event. The choice of event handler may be made in one of two ways:

- A scheduler iterates through every trigger and dispatches the related event handler if the trigger returns true;
- A scheduler examines a central trigger database consisting of mappings from events to event handlers.

Software Entity	Software Architecture Entity Type
Event	Data component
Event handler	Functional component
Trigger	Functional component
Trigger Database	Data component

Table 4 - Event-Based Architecture Software Entities

The fact that an event-based architecture is a niche architecture means that the architecture will not be adaptable for new requirements which assume a different architecture.

Event-based architectures possess limited extensibility: new events, triggers and event handlers can be added dynamically to a running event-based software system and the behaviour inherent in the new software entities can be utilised dynamically without hard-coded control constructs having to be written in order to utilise them. As Pountain et al point out, however, extensibility is typically one-way, so that new code can use old code but old code can not, in general, use new code unless hooks have been provided from the outset [Pountain94a]. The latter depends on the ability to define an interface which is generic enough to allow new software entities to implement it in the future. In this way, existing code can call an interface which is implemented by different software entities at different times. An example of this is the vnode architecture of the UNIX device architecture, which provides an abstract interface which can be implemented by different filesystems, so that one interface provides access to different filesystems without having to provide a different interface for each one.

2.5 Aspect-Oriented Models

These types of models emphasise separation of concerns and include:

- Subject-Oriented Programming;
- Intentional Programming;
- Aspect-Oriented Programming;

which were discussed in chapter 3. They all possess similar characteristics with respect to the evolveability issues which have been discussed in this thesis.

In general, a lack of both coupling and hard-coding of assumptions between aspects should result in good containment of evolution and adaptability.

As a representative example, consider subject-oriented programming.

The software entities which comprise the subject-oriented programming model are shown in Table 5. The Subject-Oriented model is built on top of an object-oriented model, which encapsulates many of the other software entities and architectural aspects such as messages.

Software Entity	Software Architecture Entity Type
Subject	Functional and Data Component
Subject Composition Operator	Functional Component

Table 5 - Subject-Oriented Programming Model

The primary contribution of the Subject-Oriented model is that of a high-level abstraction which provides a basis for separation of concerns with respect to domains. The underlying object-oriented model provides an implementation of a particular concern which can then be composed with other concerns through subject composition operators. The

flexibility of the model is limited by the flexibility of the underlying object-oriented model. Extensibility can be applied both to the underlying object-oriented model and subjects themselves. Hence, the subject model itself is potentially extensible; new subjects and subject composition operators can be added to a software system, and existing subjects removed without much effect on other parts of the software system. Removal of subject composition operators may produce ripple effects however.

2.6 Adaptive Software (AP – an Extension to Object-Oriented Models)

Software Entity	Software Architecture Entity Type
Class	Data Component
Object	Data Component
Class Method	Functional Component
Class Data Member	Data Component
Object Data Member	Data Component
Message (local procedure call semantics)	Connector
Propagation Pattern	Functional Component
Visitor Method	Functional Component

Table 6 - AP Software Entities

Adaptive software, as proposed by Lieberherr [Lieberherr96a] extends the basic object-oriented model with the notion of propagation patterns which encapsulate behaviour across a number of collaborating classes. Behaviour is expressed using propagation patterns in terms of the traversal of a class graph in such a way that the behaviour is de-coupled from the particular class structure used. For *certain kinds* of changes to the underlying class structure, the propagation patterns aren't affected i.e. those which don't alter the model in a way which invalidates the requirements made of it; for example, removal of classes.

An object-oriented model can be expressed in terms of a functional model by passing the object (instance of class) as a parameter to the method – the so-called "this" parameter. If expressed this way, Adaptive Software provides limited flexibility of behaviour, in the form of multi-class collaborative behaviour[2], with respect to changes in the structure of individual parameters, but it's still constrained by the inflexibility of the underlying object-oriented model.

The Demeter project at NorthEastern University [Lopes94a] approaches the adaptability of certain functional aspects of code with respect to the data aspects by the use of a mechanism espoused in the UNIX community; that of expressing code in terms of the most general case possible. So, the functional aspects are expressed in terms of the most general data

[2] As opposed to individual methods. Multi-class behaviour, as explored in research on contracts as well as propagation patterns, is a non-class form of software entity that expresses behaviour across a graph of classes.

structure possible, designed in such a way as to be a generalisation of any future change to the data structure. Hence, a future data structure will result in the data structure being a specialisation of the data structure in terms of which the code is written. In effect, future requirements are predicted.

2.7 Reflection and Open Implementation Models

Computational reflection and open implementation, in the way used by Kiczales et al [Kiczales91a] and Maes [Maes87a], is good at extensibility but is constrained by the extensibility of the hook interface, which is dependant on:

- The abstractness of the hook interface. The fewer assumptions about which functions make up the interface the better, and the more generic the parameters to these functions the better;
- The flexibility of the requirements which the hook interface implements. If the requirements aren't very constrained (and hence more flexible), then the hook interface is more open and extensible.

Hence, reflective models provide a hook for extension of reified entities through the meta level.

The flexibility of a design built using a reflective approach is dependent on:

- Which aspects of the base level are reified;
- The operations in the meta level.

Flexibility deals with the ease with which aspects of a design, such as design decisions, can be changed and reflection allows the implementation of base level concepts to be changed within a bounded evolution space. For example, in Maes' research [Maes87a], the implementation of messages can be changed from local, synchronous semantics to local, asynchronous semantics, and multiple inheritance can be introduced into a single-inheritance system through the reification of the inheritance abstraction. The scope of such changes is limited by the amount of reification present in a reflective model. Hence, the more reification, the more flexible a software system because more aspects of a software system can be changed more easily by re-implementing them.

However, reflective models don't address adaptability. Hence, the adaptability of a reflective model is dependent on the underlying models used for the base and meta levels. For example, Kiczales et al utilise an object-oriented model for both the base and meta levels [Kiczales91a], so that adaptability of the reflective model as a whole is governed by the adaptability of the object-oriented model.

2.8 The Relational Data Model and SQL

SQL is a type of specialised functional model consisting of specialised constructs for manipulating relational databases, such as "SELECT", "PROJECT", "JOIN" etc. Evolution of tables is governed by normal form, which aims to ensure consistency of relational data, and may result in the creation of new tables and keys. The effects of this on any SQL code is dependent on the code, because of the high dependence of SQL on any relational schemas it uses. Hence, most

changes to a relational database will affect any SQL code which uses it. There is little containment of evolution in the data model, so that the following changes may affect SQL code:

- Table name changes;
- Table attribute changes;
- Changes in the dependencies between keys and attributes;
- Changes in the keys, such as expansion (addition of attributes to a key) or contraction (removal of attributes from a key).

Containment of evolution in SQL code is dependent on the design also. A good design will hide changes in the implementation of a particular database operation behind a well-defined interface such that, in the absence of changes in requirements, changes in the implementation will not affect any SQL code using the operation.

Consider the scenario depicted in Table 7 which shows a relational model evolving from a table, T_1, consisting of five attributes, A_i, of which A_1 is the key, into tables T_1' and T_2, consisting of a shared key, A_1, and attributes A_2, A_3, A_4 and A_5. In addition, the table shows how the SQL code (which displays every tuple in the model) has to adapt in response to the changes in the relational model. SQL is not very adaptable with respect to changes in the relational model, as this example shows. The relational model models the same data, but the SQL needs to be changed in order to be able to provide the same behaviour with respect to the evolved data.

	Before Evolution	After Evolution
Relational Model	T_1 ($\underline{A_1}$, A_2, A_3, A_4, A_5)	T_1' ($\underline{A_1}$, A_2, A_3) T_2 ($\underline{A_1}$, A_4, A_5)
SQL Code	SELECT * FROM T_1	SELECT * FROM T_1, T_2 WHERE $T_1.A_1 = T_2.A_1$

Table 7 - Evolution of a Relational Model

Referential integrity imposes constraints on a particular relationship by enforcing particular rules about changes in the *values* of attributes in a table. Typical constraints include, for example:

- An employee record can't be removed from the "Employees" table if there are orders assigned to the employee in the "Orders" table;
- A primary key value in a primary table can't be changed if that record has related records. For instance, an employee's ID in the "Employees" table can't be changed if there are orders assigned to that employee in the "Orders" table.

which are typically imposed by functional components. However, referential integrity can't cope with the effects in changes in the structure of a table on other tables which are dependent on keys in the evolving table. Normal form deals only with the integrity of tables, and not with adaptability issues. Hence, the effects of changes in the primary key of a table, for example to include an extra key component, can be difficult to trace to tables which depend on the original key,

although many relational database software systems allow the user to view explicit join operations on tables. However, changes in a table may affect other tables which are implicitly dependent on the changing table due to a shared key. There is little support in the relational model for this and adaptability suffers as a result.

2.9 Summary

Table 8 and Table 9 summarise the foregoing analyses on the evolveability of existing software models.

Software Model	Feature Interaction	Adaptability	Flexibility
Demeter/Adaptive Programming (AP)	No support.	Yes – adaptability of function with respect to data. The lack of explicit dependencies between propagation patterns means that their removal should cause no ripple effects. However, this may not apply to semantic concerns of propagation pattern removal. The addition of propagation patterns may affect existing propagation patterns semantically. Visitor methods are affected by the ripple effects of changes in the class to which they are attached, just like any other method.	Same problems as standard object-oriented models.
Hursch [Hursch95a]	No support.	Yes – adaptability of data with respect to data, constrained to object-oriented models.	Same problems as standard object-oriented models.
Object-Oriented Models	No support.	Tight coupling imposed by inheritance on the super-/sub-class relationship decreases adaptability and introduces the potential for ripple effects. Changes in the class interface (consisting of messages attached to methods) can have far-reaching effects on the class structure.	• Class structure inflexibility • Method visibility inflexibility.
Functional Models	No support.	Limited support.	Limited support for revocation of design decisions.

SQL, Relational Model	No support.	Limited support for adaptability of SQL with respect to changes in the relational model. Limited support for the adaptability of tables with respect to evolution in tables on which they depend.	The lack of constraints imposed by the table abstraction on the modelling of data makes the relational model fairly flexible.
Blackboard Architectures	Not applicable, because changes are typically extension changes. The only conflicts which occur concern conflicts in pre-conditions on knowledge sources.	Not applicable, because changes are typically extension changes.	Very flexible within the confines imposed by the blackboard model because existing knowledge sources can be removed and new ones added without much effort.
Event-Based Architectures	Not applicable (see explanation for "Blackboard Architectures").	Not applicable (see explanation for "Blackboard Architectures").	Hooks are provided for the addition of new events and event handlers, providing they don't break the interfaces of the model.
Entity-Relationship Model	No support.	Dependent on the modelling requirements. Adaptability with respect to user modelling requirements is difficult because new requirements may be drastically different.	
Reflection and Open Implementation	No support.	No additional support above that provided by the underlying functional or object-oriented model.	Dependent on the amount of reification and the operations permitted on reified entities in the meta level.
Subject-Oriented Programming (SOP)	No support.	Removal of subjects shouldn't cause ripple effects because no other entities depend on them. Removal of subject composition operators may produce ripple effects on subjects which use them. Adaptability of subjects with respect to their object-oriented implementations is dependent on the subject and	Unknown.

		changes to the object-oriented implementation.	
Aspect-Oriented Programming (AOP)	Similar to "Subject-Oriented Programming", which is a form of aspect-oriented programming.		
Domain-Specific Languages	No support.	Language-dependent.	Language-dependent.
Intentional Programming (IP)	Similar to "Subject-Oriented Programming", which is a form of aspect-oriented programming.		

Table 8 - Summary of Evolveability 1

Software Model	Extensibility	Support for Localisation of Evolution
Demeter/Adaptive Programming (AP)	Same as object-oriented models, with the added complication of the existence of propagation patterns. The AP model provides no real constraints to the addition of new propagation patterns	Propagation patterns localise changes to class structure to the class model, thereby improving the adaptability of behaviour with respect to such changes.
Hursch [Hursch95a]	Same as object-oriented models.	Same as object-oriented models (no new abstractions).
Object-Oriented Models	Supported through inheritance.	Limited by the quality and stability of the design. In general, classes create more interfaces and so more localisation through cohesion.
Functional Models	No support	Dependent on the design. No inherent support.
SQL, Relational Model	No support	Limited
Blackboard	Limited support for knowledge sources expressible in terms of existing concepts used in the blackboard.	Not applicable.
Event-Based Architectures	No support	Not applicable.
Entity-Relationship Model	No support	Not applicable.
Reflection and Open Implementation	Yes, limited to extensions which don't break interfaces; constrained by the extensibility of the hook interface.	Only for changes which don't break interfaces.
Subject-Oriented Programming (SOP)	Existence of the abstractions "subject" and "subject composition operator" provides for extensibility. Addition of	Dependent on the relationship between a subject and the object-oriented model which

	subjects and subject composition operators should not cause ripple effects.	implements it. Changes within the requirements imposed by the subject will be localised. Others will not.
Aspect-Oriented Programming (AOP)	Similar to "Subject-Oriented Programming"; new aspects can be added and removed without affecting other aspects. The creation of new aspects is greatly influenced by the ability to determine how to integrate them with existing aspects.	Dependent on the aspects chosen. However, in general, choosing aspects with little dependence on each other and which make as few assumptions about each other as possible, should result in good containment of evolution.
Domain-Specific Languages	Typically difficult to extend, like most languages (with the exception of LISP, for example, which is generic enough to support a wide range of extensions without "breaking" the language).	Language-dependent.
Intentional Programming (IP)	Similar to "Subject-Oriented Programming".	

Table 9 - Summary of Evolveability 2

In conclusion, there is no existing software model which possesses all the desirable evolveability characteristics discussed in chapters 4, 5, 6 and 7, although some support particular types of evolveability to varying degrees, as Table 8 and Table 9 show.

3 A Comparison of SEvEn with Existing Models, Architectures and Languages

This section compares the evolveability of SEvEn to the software models described in the previous section.

3.1 Introduction

One of the evaluation criteria in chapter 1 is support for **re-configuration**, **integration**, **primary** and **secondary** evolution. Support for re-configuration evolution is afforded by the existence of software entities which permit the expression of sufficiently large evolution spaces by limiting assumptions and moving them into the software entity interface. These include:

- Tasks and services. Behaviour parameters permit more abstract task abstractions by allowing behavioural parameters;

- Services and software architecture. The removal of assumptions about architectural characteristics such as patterns of communication and location of services results in adaptability of services and the casting of what would have been integration evolution when software architecture changes to re-configuration evolution in which some of the decisions about software architecture are moved into the message interface.

The domain-independent characteristic of SEvEn ensures wide applicability of the approach across many domains.

This thesis contributes to a better understanding of software evolution and evolveability, through the investigation of re-configuration and integration evolution and adaptability, flexibility, adaptability and extensibility (discussed in the sections below), and how these affect software evolution, through the analysis of primary and secondary evolution.

Whilst SEvEn utilises inheritance for DEMs through *IsA* relationships (which provides a useful abstraction mechanism), it doesn't impose constraints on method visibility and encapsulation like object-oriented models.

3.2 Flexibility

The main objective of flexibility is to limit the constraints imposed by the software entity model which create an inertial barrier to evolution of software. These constraints can take many forms, but all limit the adaptability of a software system with respect to new requirements. Note that flexibility is intimately tied in with feature interaction (or conflicts between existing requirements and new requirements), because feature interaction implies changes in existing aspects of a software system, the ease of which is determined by how flexible the software model is. The flexibility of a software model is concerned with designing the model such that design decisions and constraints (which are an inevitable characteristic of any engineered system) can be easily changed, and that such constraints don't have to mean that they stay with the system for life. In SEvEn, the aim was not to aid the software engineer in determining feature interaction in a given software system and determine the evolution steps which would resolve it, but to identify approaches which could ease changes made to design decisions and constraints. The following types of flexibility have been identified:

- No constraints on the mapping between what a service wants to do and how to do it. This is achieved through the task-service separation of concerns, in which the task encapsulates what to do and the service encapsulates how to do it. The mapping is potentially dynamic within a particular evolution space, so that a set of services can be called for a given task, dependant on a set of behaviour parameters;
- No constraints on service visibility. This is in contrast to object-oriented models in which the class model constrains the visibility of method calls. SEvEn's approach is to permit a service to call any task, dependant only on the availability of data and resources. Hence, visibility of task use is determined by:

    - The data available in the current state space;
    - The set of DEM mappings available.

    In order to increase the visibility of tasks for a particular service requires changes in one or more of the above. This may simply be a case of introducing another DEM mapping, or a more complex change that involves changing task

and service interfaces to include more parameters. Hence, the SEvEn model weakens the constraints imposed on service visibility (in contrast to the class abstraction in object-oriented models) in order to improve flexibility of the model, at the cost of loss of encapsulation and cohesion. However, it is interesting to observe that object-oriented models are useful for relatively static domains (such as the user interface domain), where extensibility is the norm and inheritance can provide this. In comparison, they are not good for domains in which *existing* classes need to change, resulting in the fragile base class and related problems [Mikhajlov97a].

- Limited constraints on the choice of software architecture. SEvEn improves flexibility with respect to software architecture by allowing it to be changed more easily. This is accomplished by breaking direct dependencies of services on a particular software architecture, so that the software architecture can be changed more easily and with less of an effect on the FSEs. This is in contrast to most existing software models, especially niche models such as event-based architectures, in which changes in architecture are difficult to perform;
- Data flexibility: the DEM data model provides a flexible data model which assumes no particular structure of data, other than that it consists of data entities related by *HasA* and *IsA* relationships. Unlike current software languages, DEMs:

  - Provide a good basis for the specification of data conversions using DEM mappings;
  - Can represent most data structures in existing software languages;
  - Are not constrained to a particular structure because they are generic;
  - Have a well-defined evolution space and interface to other software entities. This is in comparison to, for example, linked lists in some current software languages whose evolution space is not linked to effects on other aspects of these languages.

  Also:

  - Changes in a DEM can be applied to any DIM instances and the effects on other software entities determined;
  - Changes in a DEM don't result in re-compilation of the software, unlike data structures in current software languages.

3.3 Adaptability

A number of types of adaptability have been described in chapters 6 and 7:

- DIM adaptability with respect to DEM evolution;
- DEM adaptability with respect to DEM evolution;
- Service adaptability with respect to service evolution;
- Service adaptability with respect to software architecture evolution. The solution to this lies in recognising the factors contributing to architecture, such as the targets of messages and patterns of communication and providing an interface for them. This allows the targets and patterns of communication to be changed without causing ripple effects in the FSEs;
- Service adaptability with respect to DEM evolution;

- Service adaptability with respect to message evolution.

In comparison to Demeter, for example, the adaptability measures in SEvEn are more far-reaching. These forms of adaptability are not present in current software models.

3.3.1 Scope and Limits of Adaptability

An important aspect of adaptability is its scope i.e. how adaptive a software entity is with respect to changes, or how many changes it can adapt to in a particular entity on which it depends. There are a number of aspects to adaptability:

- What the client requires of the server. For example, a service's behavioural requirements of those services that it uses, or the data modelling requirements of those data structures that it uses. The constraints on these requirements determine the adaptability of a client with respect to a particular server. These constraints aren't generic, but depend on particular clients and servers. The more constraints exist, the greater the probability that changes in the server will break the adaptability of the client;
- The stability of the server i.e. how prone it is to change. An algorithm, for example, is static and doesn't change because it captures the essence of the problem being solved. Other servers, however, may have many implementations. The implementation chosen may depend on factors such as speed requirements or compatibility with other entities such as hardware. Changes in these requirements may result in changes in the implementation, some of which may not be compatible with the client and hence break its adaptability. This, in turn, will result in ripple effects i.e. changes outside the interfaces hard-coded into the software.

An aspect of requirements which is of importance to evolveability is that of the close world assumption and its effects on requirements. The closed world assumption [Ginsberg94a] states that, for a set of logical assertions, any other logical assertion which can't be deduced from this set is false. Although a logic-based definition, it can be applied to any situation in which the truth of a fact, given a set of facts, needs to be determined. For example, given the set of facts:

OlderThan (Steve, John)
OlderThan (Sarah, Marie)

the closed world assumption states that:

OlderThan (Steve, Sarah)

is false because this fact is neither in the set of facts nor can be deduced from the set of facts.

The application of the closed world assumption to requirements concerns the requirements which a client entity makes of a server entity which it uses. For example, service, S, calls service "GraphLayout" in order to lay out a particular graph. Hence, S requires that "GraphLayout" lays out the graph. However, does this allow the possibility of "GraphLayout" also colouring in the graph? The closed world assumption would not permit this because it isn't explicitly specified in the requirements. However, requirements are often not fully specified and often make many

assumptions. This means that, whilst they may not explicitly specify a particular requirement, this requirement is still perfectly desirable. This is especially true of some non-functional requirements such as speed of execution, which is typically not specified directly in requirements that functions make of another function. This makes it difficult to determine if evolution of a called service breaks any non-specified requirements regarding speed of execution.

A task guarantees a particular set of abstract requirements, which any service that *Implements* the task satisfies i.e. doesn't break, contradict or extend. In this case, however, the closed world assumption is not very useful because tasks may specify abstract requirements which, by definition, do not *completely* specify the requirements. This leads on to a definition of the limits of, or constraints on, adaptability:

- The requirements which the client software entity makes of the server software entity must not change;
- The *implementation* of the server may change within the evolution space defined by the requirements, and encapsulated in the server software entity.

In summary, adaptability is successful only if the following conditions are met:

1. The client's requirements of the server are stable/static throughout the evolution;
2. The server's evolution doesn't break the requirements made of it by the client.

Chapters 6 and 7 have identified how the adaptability of clients can be improved with respect to servers in light of these adaptability conditions.

3.4 Extensibility

Any software system is extensible to some degree. Some are more extensible than others; for example, object-oriented systems, through inheritance, provide a well-defined interface that allows aspects of the code to be extended in well-defined ways. Similarly, approaches using reflection, such as Maes [Maes87a], provide a so-called meta-level for making changes to a base level. The meta-level provides a well-defined extension interface for base level concepts. Even functional software models permit new functions and data structures to be added to a software system, but their extensibility is let down by an inherent lack of an *interface* for integrating such extension changes.

A primary characteristic of extensibility is the ability to dynamically add new capabilities to a running software system. A major constraint on extensibility is that new capabilities must conform to existing interfaces in the software. Otherwise, ripple effects will occur.

The task-service separation of concerns permits limited, two-way extensibility (pre-evolution code can call new code and new code can call pre-evolution code) [Pountain94a] by providing an architectural means for existing services to call new services. Much like the streams and device framework in UNIX provides an API which presents a common, un-changing interface to clients and allows different implementations to be created which *Implement* the API, the task-service separation of concerns allows a software engineer to create a task interface which provides a common interface

for which different service implementations can be created. In this way, software can be extended by creating new services which *Implement* tasks and don't break interfaces (it is also conceivable that new tasks which *Implement* existing services could be created, although this would probably be more difficult). This is, however, limited extensibility.

Hence, there are two types of extensibility:

1. An existing software entity *Uses* a new software entity;
2. A new software entity *Uses* an existing software entity.

Uses of both is two-extensibility. 2 is easier and 1 is more difficult unless interfaces already exist to allow it to take place as provided by, for example, the plug-in architecture of Netscape and the "vnode" and device-driver architectures of Solaris.

3.5 Level of Help in Determining the Effects of Change and Ripple Effect Management

As discussed in chapter 2, software evolution can be divided into primary and secondary evolution. Primary evolution is concerned with adding new software entities to a software system which don't conflict with existing software entities and hence produce no ripple effects. Secondary evolution, in comparison, produces ripple effects which result from conflicts between new and existing software entities in a software system. This section is concerned with secondary evolution, the resultant ripple effects and how the current situation can be improved upon.

This is achieved through the reflective software entity model, which provides a way of modelling the software entities in a software system and their inter-relationships. In addition, it provides a way to model:

- The evolution space of a software entity;
- The aspects of a client software entity's *server* which may be affected by its evolution.

In this way, changes in a software entity acting as a server can be mapped to effects on the servers characteristics which can, in turn, be used to determine any effects on the servers clients, in situations where client adaptability has failed.

Where assumptions can't be extracted out of software and the adaptability and flexibility improved, ripple effects are inevitable. Ripple effects are caused by assumptions implicit in dependencies between software entities. Both assumptions and dependencies can be implicit (i.e. unknown until they affect the software in some way), but must be made explicit in order to determine how changes in a software entity can affect other software entities which are dependent on the changed software entity. In current software models, ripple effects may not become apparent until the software doesn't behave the way its specification says it should. This is because a change has been made to a piece of software which breaks an assumption (such as an interface which is not explicitly modelled in the software). For example, a change to a function may alter the speed of that function. Another function which uses this function may

assume a particular speed requirement of the function. In this case, the change in the function will affect the other function. However, because the speed requirement isn't explicitly modelled as part of an interface such a change will have an undetermined ripple effect on the other function. Hence, there is a need for software engineers to model as many of the requirements and assumptions that software entities make of those software entities on which they depend as possible, so that in the absence of inherent adaptability ripple effects are made explicit and easily determinable. Again, total ripple effect coverage is not possible because not all assumptions and requirements can be determined by the software engineer (this has been discussed elsewhere but, in short, software engineers may not know they're making assumptions and some requirements may be implicit in their thinking and subsequent design).

Existing approaches to ripple effect analysis depend on probabilistic methods or expert judgement, and identify only *what* changes. SEvEn aims to ease ripple effects by:

- Identifying where they will occur. This is accomplished by employing the use of appropriate information modelling using the software entity modelling framework described in chapter 5. This information takes the form of:
  - Improved dependency information between software entities;
  - Improved removal of assumptions, if possible. If this isn't possible, the explicit modelling of assumptions so that the effects of changes on assumptions which cause ripple effects, can be traced more easily. Obviously, all assumptions can't be determined, but certainly more assumptions can be modelled than is currently the case;
- Identifying *how* changes in a software entity affect other dependent software entities.

Complete adaptability is not possible for a number of reasons:

- Unpredictability of changes: if all future changes could be predicted and clients designed in order to be inherently adaptable to all future changes, then ripple effects wouldn't occur. However, complete predictability is impossible and therefore clients can not be made adaptable to all potential changes;
- Conflicts between new requirements and existing requirements implemented by the code will inevitably affect the code, no matter how flexible it is with regard to changes.

Hence, particular kinds of entity evolution (be they to requirements or software entities) will cause ripple effects that will percolate to clients that are not adaptable to the changes occurring in the entities on which they depend, as shown in Figure 4.

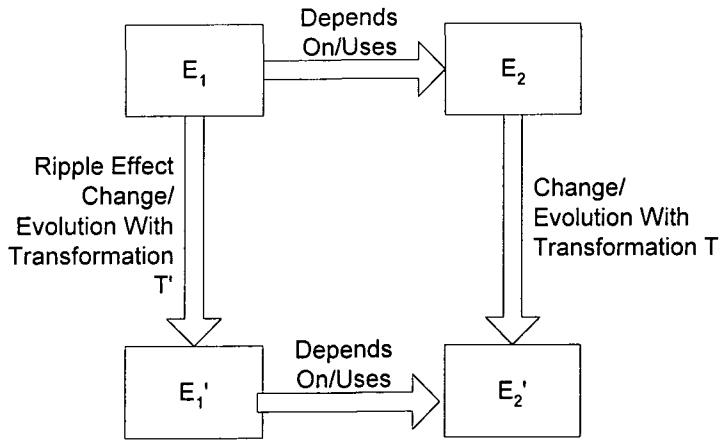


Figure 4 - Ripple Effects and Evolution

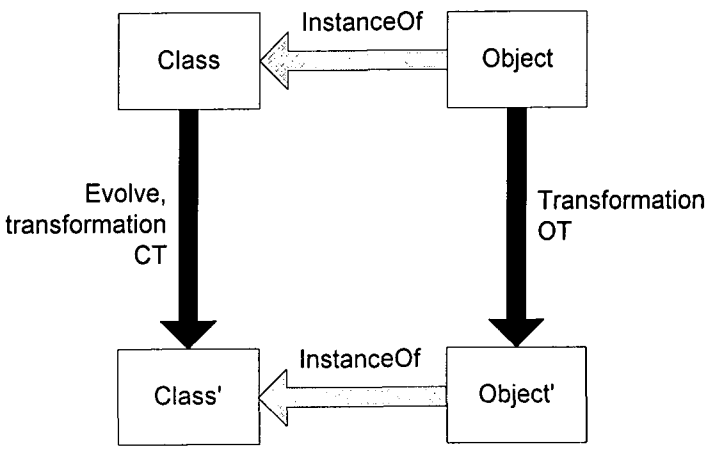


Figure 5 - Adapted from [Hursch95a]

There are two types of trigger for a ripple effect:

- A change in a client affects a server e.g. a client service, S_C, passes an extra parameter to server service, S_S;
- A change in a server affects a client e.g. server service, S_S, has an extra parameter added which all client services must pass to S_S. If these clients don't already do this (which will probably be the case), then they will have to adapt to this ripple effect.

In addition, there are two factors that influence the *type* of ripple effect that occurs between two entities in a computer system. In the context of the model presented in chapter 5, these are:

- The type of relationship between the two software entities;
- The types of the two software entities.

Together, these prescribe the type of ripple effect between the two software entities concerned. So, for example, the types of ripple effect that occur between a functional software entity and a data software entity are different than the types of ripple effect that occur between a data software entity and a requirement.

Hursch, in [Hursch95a], develops a framework for maintaining consistency in object-oriented software by looking at how changes in the class model break other parts of the class model. For example, how does removing a superclass affect other parts of a class model? Hursch enumerates a set of class model changes and then proceeds to determine the effects each change can have on the rest of the class model. The approach is at the other end of the spectrum when compared to adaptive software approaches, because it employs the use of transformations applied to clients in response to changes in those object-oriented model software entities on which they depend.

As long as all dependencies are modelled, it should be possible to catch ripple effects that occur as a result of the in-adaptability of clients. However, not all dependencies are typically modelled within software systems. Take the example DEM in Figure 6. Each edge consists of two nodes, "Node1" and "Node2", which are both integers and index into the sequence of nodes. Hence, there is an implicit dependency between edges and nodes through node instance numbers. These implicit types of dependencies are difficult to extract from code and it is difficult to determine ripple effects through such implicit dependencies.

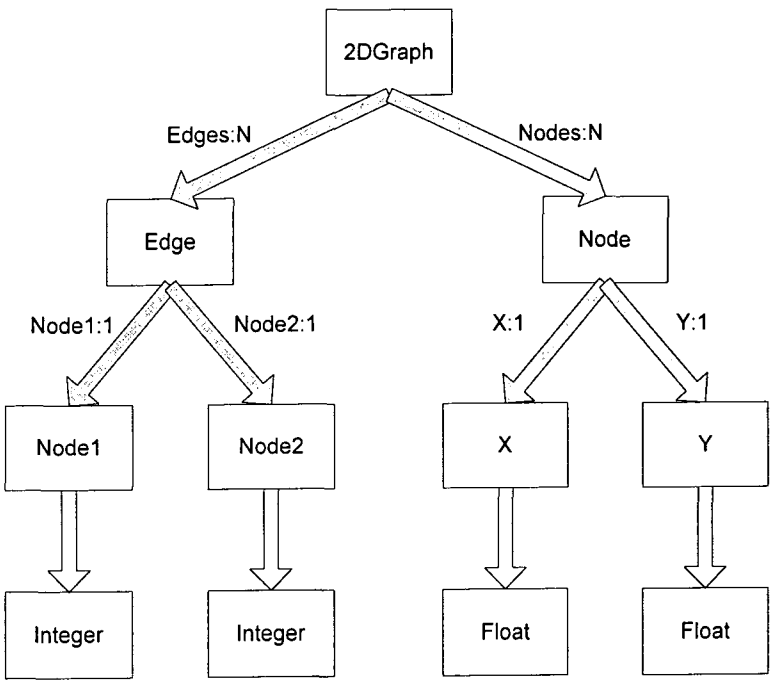


Figure 6 - $DEM_{2DGraph}$

Within the context of the two types of evolution identified in chapter 2 (namely re-configuration evolution and integration evolution) and for the purposes of this thesis, a ripple effect can occur as a result of adaptation/integration evolution but not in re-configuration evolution because its definition assumes that any software entities required by the change are part of the extension of the software. An evolution change in a software entity may affect any other software entities in its environment. As described in chapter 4, an evolution change is a change within an evolution space, defined by:

- Add;
- Remove;
- Change/refine

operations on aspects of a software entity.

3.6 Localisation of Software Evolution

Existing software languages, architectures and languages contain very few abstractions and, as a result, few interfaces. This means that evolution is not very localised and the targets of evolution difficult to determine. SEvEn consists of more abstractions (software entities), which both localises evolution behind the interfaces produced and provides more targets for evolution. For example, functions in current software languages typically encapsulate implicit data conversions so that heterogeneous data can be converted into a form useable by the function. The implicit nature of these data conversions makes it difficult to evolve them and determine their interface to other aspects of the software. SEvEn provides an explicit data conversion abstraction, the DEM mapping software entity, with well-defined interfaces to other software entities in SEvEn and better-defined evolution.

SEvEn provides the following increases in interface:

- Task-service. In traditional software, what to do and how to do it are both encapsulated in the function abstraction. The separation of these two different kinds of information into different abstractions task and service provides an interface which localises particular kinds of change to just the service abstraction;
- Brokers;
- DEM Paths and DIM Paths.

In existing software models, the parsing machinery is essentially part of the function, which is then affected when changes in the data affect the parsing. This then affects the function itself, which isn't helped by the fact that the parsing machinery may be entangled with other aspects of the function.

4 Summary, Discussion and Conclusions

In conclusion, no existing software model provides a set of software entities which are completely evolveable. Section 2 has shown this for a number of existing software models. However, the evolveability of these software models can be improved, as has been shown with SEvEn. Ripple effect analysis has also been improved by identifying both:

- Which software entities have to change in response to a change in another software entity;
- How these software entities change.

by relating change types to their effects on the software entity interface.

In addition:

- Ripple effects are difficult to determine because of the inadequacy of interfaces in existing software languages, models and architectures. This results in "contracts" between client and server entities in a model failing to capture

all characteristics of the contract. For example, function interfaces fail to capture certain characteristics of the function which may be affected by changes in the function, such as duration, speed and ability to satisfy requirements. Ripple effects can't be avoided, but the inadequacy of interfaces fails to recognise that ripple effects are occurring as a result of changes "behind" the interface;

- Inability to localise evolution in certain situations for current software models. For example:
  - Changes in architecture affecting software entities because of the assumptions the software entities make about architecture;
  - Changes in the mapping between what to do and how to do it breaking the function which is requesting something to be done on its behalf when they shouldn't i.e. changes in this type of mapping shouldn't break the calling service.

The main conclusions, which are expanded on in chapter 9, are that complete evolveability is not possible and ripple effects will always be a problem. A potentially useful approach which would go further in dealing with the problems of ripple effects caused by conflicts in requirements would be to be able to link requirements to aspects of the code which implement the requirements, coupled with an ability to determine requirements conflicts and revoke existing requirements. Finkelstein et al have been working on the area of requirements conflicts for some years [Finkelstein90a] [Finkelstein92a] [Finkelstein94a], whilst Karakostas, for example, has done some work on the linking of requirements to code [Karakostas90a].

Chapter 9

Results, Conclusions and Further Work

1. Results and Discussion

Like other researchers [Lopes94a, Alexandridis86a, Kishimoto95a], this thesis has identified software adaptability (and the encompassing evolveability) as an important characteristic of software. There are three particular aspects of software evolveability: **flexibility, adaptability** and **extensibility**. This thesis has presented a number of approaches to improving the evolveability of software, through the development of a software entity model which consists of a set of software entities (abstractions) and their inter-relationships. The aims of this software entity model are two-fold:

- To provide a set of abstractions that build on existing abstractions in programming languages, software models and architectures and that software engineers can use to create software systems;
- To increase the evolveability (adaptability, flexibility, extensibility and localisation of evolution) characteristics of software systems developed using the model.

Two broad, orthogonal *taxonomies* of software evolution have been identified:

- Re-configuration and integration evolution:
  - Integration evolution, in which new capabilities are required that must be integrated with existing capabilities;
  - Re-configuration evolution, in which changes are to existing instances of abstractions/software entities;
- Primary and secondary evolution:
  - Primary evolution is characterised by requirements which result in *extensions* to an existing software system;
  - In contrast, secondary evolution is characterised by new requirements which *conflict* with existing requirements and design decisions in a software system and result in ripple effects.

Re-configuration evolution is arguably easier than integration evolution because the changes have essentially been predicted and measures built in to overcome the difficulties they pose through redundancy. The real problems for evolution lie with integration evolution. Current approaches, however, seem to favour re-configuration approaches to software evolution. Many of the current approaches to improving the evolveability of software are based on open implementation techniques, whereby the implementation of particular aspects of the software is decided at run-time. At design time, a space of potential implementations is designed. Hence,

software evolution is changed from integration type evolution to re-configuration evolution. The main problems with these approaches are:

- All implementations must be compatible with one interface and;
- The limits of evolution are decided by the number of implementations. Any changes outside the **evolution space** provided by the open implementations result in integration evolution.

A major problem posed by software evolution, however, is when changes occur *outside* the existing interfaces in a software system. This can potentially produce ripple effects, because software entities which are dependent on the changing interfaces must be adapted. This thesis has shown how an increased set of abstractions, in turn, creates more interfaces and thereby improves the localisation of software evolution. Many more changes are essentially hidden behind the increased number of interfaces which are created. The inherent adaptability of software entities can also be improved, an approach which limits the number of assumptions that a software entity makes of other software entities on which it depends. This means that when a software entity changes, there is less chance of the change breaking assumptions that dependent software entities make. These two approaches to improving the evolveability of software are not enough to prevent all problems associated with software evolution and ripple effects, for a number of reasons:

- All assumptions can't be removed, because a software engineer can make assumptions without being aware of making them, and assumptions are the direct result of design decisions which are an essential feature of any design. A completely unconstrained design is impossible because trade-offs must be made and representations utilised;
- There will always be changes that can't be contained within an abstraction or interface and so will inevitably break the model. One of the aims of this thesis is to *improve* this aspect of software, with the assumption that it is either impossible or very difficult to completely eradicate the problem.

Adaptability is concerned with how well a set of software entities comprising a software system or sub-system adapt to changes in their environment (consisting of entities - software or otherwise - on which they depend and which may depend on them). A high level of adaptability indicates that the software entities are able to cope with many changes in their environment, without having to change much themselves. This is achieved on a software entity by software entity basis by limiting the assumptions that software entities make about their environment.

Whereas adaptability deals with the ability of a software entity to *overcome* changes in its environment, flexibility is concerned with how easy it is to make changes in response to changes in requirements. Hence, if a software entity is found not to be adaptable to a particular change in its environment, flexibility measures how easy it is to evolve the software entity. So, adaptability is concerned primarily with *limiting* assumptions and flexibility is concerned primarily with *overcoming* assumptions. A classic example of adaptability is the adaptability of function with respect to data. However, whereas the approach of Demeter is to utilise propagation patterns [Lopes94a], this thesis has opted for the use of DEM Paths.

An important characteristic of software evolveability is flexibility, or adaptability of a software system as a whole with respect to changes in requirements. This is an important characteristic particularly for requirements that conflict in some way because flexibility deals with how easy it is to adapt the software to possibly conflicting requirements changes. For example, a particular set of requirements produces a particular software configuration. A new set of requirements, as discussed in sections 5.1 and 5.4 in chapter 2, can either extend, reduce, or conflict with the requirements set. This means that the existing configuration can potentially conflict with the new requirements. For instance, the original requirements may have resulted in a particular object-oriented model which is incompatible with the new requirements, which require a different object-oriented model. This is unavoidable because, as described earlier, a set of requirements result in a constrained design, one aspect of which is the set of constraints imposed on the configuration of the software. Flexibility, however, aims to ease the process of making such changes by limiting as many design constraints as possible, such as service visibility (discussed in chapter 7 section 3.6).

Most existing software models limit the flexibility of the software, which is most important during the early stages of software development when software tends to change quite frequently. Even during software evolution, changes in requirements can result in large changes in the structure of software, which current architectures are unable to cope with. These problems stem from a number of identified characteristics of current software development:

- Implicit assumptions stemming from the fact that, for E-type applications [Lehman85b], a software system is necessarily a finite model of an infinite real-world problem and the resultant gap is bridged by assumptions [Lehman98a]. This leads to lack of adaptability of software entities;
- Inflexibility, due to built-in constraints that provide for other characteristics such as abstraction and encapsulation; for example, object-orientation improves abstraction and encapsulation, but limits the visibility of method calls and inevitably results in re-design.

An example of flexibility is the ease with which a function can be called: object-oriented models typically limit the visibility of method calls to the detriment of changes that require invalid method calls to become valid method calls. In addition, object-oriented models are inflexible and thereby difficult to change because of inheritance relationships and encapsulation and the structure they impose on software designs.

Evolveability, flexibility, adaptability and extensibility are difficult characteristics to quantify, although heuristic qualitative measures may be based on subjective measures. This thesis has not attempted to develop a quantitative theory of evolveability, but instead to look at individual aspects of evolveability and improve them with respect to current software architectures and models. Hence, a primary objective of this thesis has been to increase the flexibility and adaptability of software entities, by improving the ease with which changes can be made to software entities, and the ease with which software entities can be made to adapt to changes in other software entities on which they depend. Current software architectures and models are more geared towards easing software development, and consequently fail to overcome the difficulties posed by software evolution, although the proponents of such models extol their maintainability and evolveability without evidence to back up

these claims. In comparison, the evaluation in chapter 8 of the ideas expressed in this thesis is based on examples of how these ideas improve upon existing programming languages, models and architectures with respect to evolveability.

The research approach has been based on a process of improvement of existing software architectures, models and languages, using the results from an iterative analysis of their evolveability, that consisted of trying out changes on these architectures and determining why they're difficult to make. In software architecture terms, these software entities correspond to types of software architecture components and connectors[1] (using Garlan and Shaw's terminology [Shaw96a]). This process produced a number of observations regarding the evolveability of software:

- Increasing the modularity of software through the identification of an increased number of software entities which both build on existing abstractions and improve evolveability, such as:

  - **DEM mappings**, or how to convert data between two domains (for example, how graph data can be represented as diagram data);
  - Reification of messages. Messages are typically implicit in existing software, making them difficult to change;
  - Separation of **task** information (*what* to do) from **service** information (*how* to do it);
  - The explicit modelling of software architectural characteristics, such as location of services, types of components and connectors, and patterns of communication.

  Increased modularity has two main advantages:

  - Localisation of evolution: increased modularity implies an increase in abstraction, which in turn implies the use of interfaces, so that the effects of particular changes are restricted to *behind* the interface;
  - Adaptability of software entities can be improved more easily, because the assumptions that software entities make of their environment aren't inter-mingled with other software entities. The representation of each software entity as an independent entity means that assumptions can be more directly related to the software entity.

  Software entities are dependent on each other, and each software entity in such a dependency has a role of either client or server. Changes in the client may invalidate the server, and changes in the server may invalidate the client. For example, service A calls service B with a set of actual parameters. Service A may change the actual parameters, invalidating B, or B may change its implementation, invalidating the actual parameters sent from A. Another example is when the implementation of a task changes. In traditional software languages, the mapping between task and implementation is hard-coded. In the approach described in this thesis, the mapping can be brokered (or centralised) and changed easily. This applies to any level of

[1] Where connectors are treated as first class entities through reification.

abstraction too, so that a major change in the implementation of the task can be carried out without worrying about the existing service and how this is connected to the task;

- Limiting assumptions. Not all assumptions can be determined. Even software developers aren't able to determine if and when they're making assumptions. The key lies in determining as many types of assumption made as possible and limiting these assumptions through the use of particular design techniques. For example, assumptions about:

  - The structure of data;
  - The types of messages e.g. synchronous, local etc.;
  - The implementation of a particular task;
  - The location of a particular software entity;
  - The type of software architecture used;
  - Behavioural characteristics of a service;

- Building in reflective models of the software, which model the dependencies between software entities. Complete adaptability is unattainable, which means that ripple effects will consequently occur. Since ripple effects occur between software entities which are dependent on each other, this explicit reflective modelling can help to determine *how* changes in a software entity affect other software entities which depend on it.

  The main advantage of increased reflectivity is tied in with the argument for increased modularity, in the sense that both reflection and increased modularity provide an interface for changing the semantics of what is being reflected upon, through a well defined interface.

Chapter 5 introduced a set of software entities and their inter-relationships which form a reflective software entity model of software. Their choice was based on a number of iterative case studies which aimed to:

- Provide a set of generic, domain-independent abstractions for constructing software systems. These software entities must provide at least as much modelling power as existing software architectures and models, without imposing unnecessary constraints on the modelling process. In particular, they must allow particular patterns of architecture, such as event-based or blackboard-based architectures to be expressed in terms of them;
- Develop a software modelling framework that provides the basis for improved:
  - Localisation of evolution;
  - Adaptability;
  - Flexibility, and;
  - Ripple effect analysis.

Chapters 6 and 7 analysed how these software entities evolve, providing a set of change types for each software entity based on the concept of an evolution space [Cazzola97a]. The aim has been to circumscribe the evolution space of these software entities and relate these change types to their effects on dependent software entities, in

the absence of the ability of a client to adapt to changes in a server. An example of a lack of adaptability is that of the inability of DIMs to adapt to particular changes in the DEM on which they depend.

2. Contributions

The main contributions of this thesis are:

- The development of a conceptual framework or architectural "harness" for increased evolveability (adaptability, flexibility, extensibility, localisation of evolution), through the construction of a software entity model. The main results of this are an improved adaptability of:

  - Services with respect to DEMs;
  - Services with respect to software architecture;
  - DEMs with respect to DEMs;
  - DEM mappings with respect to DEMs;
  - DIMs with respect to DEMs;
  - Services with respect to actual parameters in the form of DIMs;
  - Services with respect to services;

  and an improvement in flexibility through:

  - Improved service visibility;
  - DEM flexibility;

  and an improvement in extensibility through:

  - The task-service separation of concerns, which provides an interface for behavioural extensibility much like polymorphism in object-oriented languages;

  and, finally, an improvement in localisation of evolution through the increased number of interfaces created by the software entities;
- Where adaptability is not possible, an analysis of ripple effect types and a framework for detecting when they may occur, as a result of assumption conflict. This has been through a reflective model that models the majority of dependencies between software entities in a software system and allows changes in a software entity to be linked to particular types of ripple effects through the effects on the software entity's interface;
- The development of a better understanding of software evolution and software evolveability, by identifying what makes software evolveable and what doesn't, and some of the characteristics of software that make software evolution difficult to perform;
- An analysis and better understanding of the limits of in-built adaptability and flexibility, constrained by the number of assumptions that can be extracted out of software;

- SEvEn improves trace-ability of changes and ripple effects and provides an architectural framework for code to self-document the dependencies which exists between software entities. However, SEvEn is no substitute for good design and, in particular, good use of encapsulation and APIs where appropriate because assumptions can't be completely eradicated. SEvEn does, however, improve on certain aspects of evolveability;
- The SEvEn model aims to be as generic and extensible as possible. For example, modelling a blackboard architecture using the SEvEn model should be as simple as adding trigger- and pre-conditions to service software entities and providing a blackboard implemented as a service instead of the default interpreter;
- Lastly, it is hoped that this thesis has described a novel use of reflection in software; that of using reflective models to represent characteristics of software entities which are affected by evolution of these software entities, and which can be used to determine the effects of such evolution on other software entities in a software system.

3. Conclusions

The main conclusions of this thesis are:

- Total inherent flexibility is not achievable within the context of software entities in existing software architectures and the software entities introduced in chapter 5. However, a hypothesis is proposed that states that, in addition to determining which software entities are affected by a change in another software entity, how that software entity is affected and must change can also be determined;
- An analysis of how software can be made more adaptable and what the limits of adaptability are for the particular software entities. For example, conflicts in requirements result in lack of adaptability, as discussed in chapter 4;
- Design for evolution is an important approach for easing software evolution and, in particular, software evolveability. Current software development methods are aimed primarily at easing software development but fail to address the problems posed by software evolution, such as changes in existing abstractions in a software system and the effects of this on interfaces;
- Whilst adaptability can be improved upon, it can't be completely achieved because new requirements may conflict with existing requirements;
- Software is only as evolveable as the models that the software engineer has put into the software allow i.e. there is a limit on the evolveability of the software that is supported by the built-in models of the software. Within these limits, the built-in models ease software evolution, but outside these limits evolution will be just as difficult as it is now. This is unavoidable since it can't be proven that all evolution types have been identified. Of course, it is hoped that all evolution types have been catered for, but there is the possibility that the set of evolution types is incomplete. However, the evolution types are based on the evolution of identified software entities, which are assumed to be complete (i.e. they can be used to model anything, as the Turing thesis states). Hence, the only possible area of incompleteness is a potential lack of completeness of evolution types for each software entity.

The abstractions in existing programming languages, models and architectures are neither sufficient in number nor sufficiently well-modelled. This results in a lack of localisation of evolution (because changes that should not affect a particular aspect do so because the aspect is tightly-coupled to an aspect which is affected by a change) and an inability to determine ripple effects (because of a lack of understanding of how changes affect the interfaces of abstractions), respectively;

- The use of interfaces with late binding and abstractions (late binding of code to architecture or data, for example) allows improved adaptability. Late binding allows the choice of server to be changed, improving flexibility, and improved interfaces provide a "buffer" that localise changes. The limitations are that the improved interface must not be broken. If it is, the software engineer must resort to traditional ripple effect analysis.
- The lack of a reflective capability in current software models. That is, a way for the software to explicitly determine the software entities and relationships of which it consists. Existing software models typically represent such information implicitly. For example, the fact that a function calls another function in C is represented within the code implicitly, and the code has no access to such information at run-time. In addition, existing abstraction interfaces provide only a limited view or interface to the abstraction. For example, functional abstraction interfaces typically consist of just data parameters. Other characteristics of the functional abstraction of importance to any clients of the functional abstraction, such as:

  - Duration;
  - Space efficiency;
  - Behaviour;

  are not represented in the interface, which makes it difficult to determine if changes in the functional abstraction affect the interface to its clients. For example, does evolving a functional abstraction change the essence of the abstraction by affecting these characteristics? This thesis has provided a mechanism for representing such information. In addition, it has produced a set of software entities which possess reflective information such as that above and allow the effects of changes on dependants to be determined. This is in addition to improving the adaptability of clients, where possible, so that changes in servers have less of an effect;
- High context dependence implies low adaptability. For example, class inheritance and *InstanceOf* relationships typically create a high context dependence and, hence, low adaptability;
- Extensibility doesn't support requirements conflicts very well, because it assumes that existing entities won't be changed. Hence the need for adaptability and, in the absence of this, ripple effect analysis. In addition, support for extensibility is largely domain-dependent. Although mechanisms for extending software entities are useful (such as inheritance in object-oriented models and languages), extensibility primarily depends on the way the system has been modelled, and on the following characteristic of the model: sufficiently abstract, reusable and context independent software entities which make few assumptions. The main point is that the domain abstractions (which are built using the domain independent software entity abstractions in this thesis) must possess the above characteristics. For this reason, this thesis has largely ignored

extensibility because it is concerned with domain-independent abstractions/software entities. Extensibility is also a feature of the domain. For example, user interfaces are extensible because the concepts they are built around are reusable in a number of contexts i.e. they are more context independent than their less extensible counterparts;

- Increasing the number of abstractions aids the containment and localisation of evolution, as well as providing more targets for evolution;
- The main problem with existing software languages, models and architectures is that they don't *directly* support the evolveability measures discussed in chapters 4, 5, 6 and 7, although such measures can be implemented in terms of them;
- A lot of the evolveability issues discussed in chapters 4, 5, 6 and 7 are dependent on the software engineer producing good designs using the SEvEn model i.e. conforming to the rules and heuristics of the model. No amount of refining of the SEvEn model can help because it depends on the software engineer ensuring a good mapping between requirements and the model so that assumptions which break contracts between software entities (realised in terms of relationships) don't creep in. In other words, the software engineer must ensure that the relationships are not broken by the design;
- It's important to determine the characteristics of changes and link them to types of change. For example, the type of change "Add a new data entity to a DEM" can be characterised by whether it affects any existing DEM paths, whether it's an extension of the DEM or results in a reduction in the modelling power of the DEM. This information can then be used to determine *types* of ripple effects.

4. Limitations of SEvEn

- Improving the adaptability of software entities at the client side of relationships aims to reduce ripple effects, but can't hope to completely eradicate them because of assumptions which can't be extracted out and new requirements conflicting with existing requirements present in the software. In this case, increased localisation of evolution at the server side of relationships will hopefully limit the effects of evolution, where these effects needn't spill over and affect other software entities. For example, changes in software architecture needn't affect other aspects of the software unless they break requirements which these aspects encapsulate. However, even increased localisation of evolution will not prevent all ripple effects, because of conflicts in requirements, assumptions and design decisions. This is where the reflective software entity model comes in. By explicitly representing all dependencies between software entities in addition to the type of dependency, one can identify when and how a change the child software entity will affect its dependent client software entities. It is also important to get the software entity interfaces right, since it is the interfaces on which clients ultimately depend, and it is changes in interfaces which have the ability to cause ripple effects. However, although the resultant software entity model allows ripple effect types to be determined, it doesn't deal with how clients can adapt to ripple effects. This is a difficult problem because it may require extra functionality or data models not present in the software, and which therefore require the intervention of software engineers. Hence, this is outside the scope of this thesis. In addition, the thesis doesn't assume that all ripple effect types have been determined;

- This thesis hasn't looked at the impact of feature interaction (conflicts between new requirements and existing requirements) on software evolution. This is a big and important area because a lot of software evolution will stem from *new* requirements which conflict with:

  - Each other (although these sorts of conflict should be resolved *before* implementation);
  - Existing requirements built into the software;

- A major problem not addressed in this thesis is that of changes in software entities (or abstractions) which result in a change in the identity of the software. These types of change are all too common in software evolution and typically result in ripple effects because they break the assumptions made by dependent software entities. Often, they cannot be avoided because the adaptability of the dependent software entities was not defined well enough, or was not understood well enough to account for such changes. This thesis has hopefully presented an approach that alleviates these problems by increasing the number of software entities and, by implication, the number of interfaces in order to improve the localisation of evolution. The argument is that, if more interfaces exist, they will be able to hide more of the changes that would normally break an existing interface and thereby affect any dependants. Hopefully, chapter 8 has shown this to be true in a number of cases where, traditionally, changes would break dependants;
- SEvEn doesn't describe how to identify integration evolution to re-configuration evolution mapping opportunities (in order to create a parameterised evolution space) because there are no generic rules or heuristics for this (although domain heuristics may exist). It does, however, provide a way of describing such transformations so that it's clear what assumptions are being made with respect to the chosen behaviour;
- SEvEn doesn't address changes in interfaces and how the rest of the software should evolve, because there are no generic rules or heuristics on how to do this. This type of knowledge is typically domain-specific;
- The thesis has necessarily been theoretical (characterised by the use of small case studies and examples, with inherent lack of scale) because of the "newness" of some of the ideas. It is probably important to try these ideas out on bigger examples and case studies;
- The identification of higher level software entities has failed because they are too volatile; the knowledge that they capture isn't stable across domains and therefore can't be utilised in a domain-independent manner. For example, in the telephone domain, it would be useful to find an abstraction (or model or pattern) that captures the similarities between members of the domain such as the operations connect, disconnect, putonhold, takeoffhold, much like the function abstraction captures the similarities between functions such as formal parameters and the use of function calls. However, members of a domain may not share similarities which allow such abstractions to be formed and, even if they did, such abstractions would be domain-dependent. The only similarities are fairly abstract and limited so that capturing them doesn't provide any real advantage. For example, all functions of a telephone switch share the following similarities:

  - They respond to a telephone line signal, and;
  - Update internal switch data structures.

  These similarities are limited and difficult to capture in an abstraction that encapsulates the similarities of switch functions. Perhaps there exist higher level software entities within domains that aren't so volatile and

are worth the effort of capturing (as Arango points out [Arango91b]). However, not all domains themselves are stable (for example, the telecommunications domain) and so higher level software entities in these domains will not be stable either. Hence, we are back to square one. This means that we must stick with existing fairly high-level, domain-independent software entities such as functions and data structures because they are high-level enough to capture similarities across domains. High-level, in this context, means that a software entity doesn't make many assumptions about what it is modelling;

- The approach described here is not going to help in planning the changes required for a new requirement, especially when the new requirement is large. In other words, it is up to the software engineers to move from an abstract high-level requirement such as "introduce billing into this telecommunications network" to a set of change requests to the code. This process will require a lot of further requirements decisions such as "should a user be billed if they have been put on hold";
- It may be difficult to determine the task-service abstractions which are relevant. The more abstract a task is the better, because then more possibilities exist for different services that implement the task, a philosophy similar to the use of more generic components advocated by the "Inventors' Paradox" pattern [Lieberherr95a] and the Demeter project [Lopes94a].

5. Further Work

Potential avenues of further work include:

- Flexibility of software as a whole with respect to new requirements, possibly through the identification of software entities which map requirements to code, so that requirements can be easily revoked. In addition, a way to "merge" requirements: given a set of existing requirements and a set of new requirements, create a merged set of requirements. This would require a method of determining whether requirements conflict, an area of work that has been pursued at Imperial College [Finkelstein94a]. Requirements conflict is a difficult problem and requires a common ontology for detecting conflicts. For example, consider the handling of errors. Function F_1 assumes that error codes will be returned as the result of the execution of called functions, whereas function F_2 (which is called by F_1) stores error codes in a global variable, which results in a conflict of requirements. The mapping of requirements to their implementation is inspired by the work of Karakostas, who describes a system in which high-level task-oriented aspects of the software are linked to the computational aspects that implement them in order to provide trace-ability of requirements to code [Karakostas90a]. An important aspect of this work would be the determination of requirements and behaviour conflicts which, when coupled with a link from requirements to code implementing the requirements, would allow the software engineer to revoke requirements directly and replace them with code which implements the new requirements. Such an approach could be based on types of requirements change: requirements extension, removal and change, in which changes in requirements are related to requirements conflicts i.e. determining what in the changed requirements conflicts with the existing requirements. This may eradicate ripple effects and the need for adaptability because conflicts are moved to the requirements model. Further work is required to determine if design decision conflicts can be dealt with at the requirements model level;

- Further investigation into assumption types i.e. the types of assumptions that are made in software and a method of extracting them out as parameters in order to move integration evolution to re-configuration evolution. The goal should be to identify individual assumptions that cover a wide range of cases, so that the extraction of an assumption has as large an effect as possible;
- A potentially useful area of work would be to look at patterns of adaptability, which could proceed by determining code patterns which make assumptions and providing solutions, where appropriate and possible, which improve the adaptability of the code. An example of this is assumptions about data structure; functions assuming that strings are terminated with ASCII 0, or iterators which assume a particular data structure such as a linked list when the underlying data structure is an array (the task-service separation of concerns could be used to provide an iterator interface, so that the task aspects of the iterator are separated from its underlying implementation);
- Investigate how to increase the number of static (unchanged by changes in the server) interfaces. This implies increasing the number of software entities. The approach is to determine client-server relationships through case studies, which will lead on to the identification of interfaces;
- Investigate how interfaces change during software evolution. A potential approach would be to perform a number of case studies analysing what happens to interfaces before and after evolution, in terms of addition of, removal of and changes in interface elements;
- Try out the ideas discussed in this thesis on bigger case studies;
- Identification and analysis of the adaptability of other (domain-specific) software entities;
- More work is required on the modelling of DEM and FSE semantics in order to aid automated DEM mapping and ripple effect analysis, respectively;
- There is a need to look at the effects of the service parameter architecture (consisting of Produces, Uses, Updates and Uses relationships and process state spaces) on remote message-passing i.e. how can these types of relationship be mapped to remote procedure calls?

Bibliography

[Abowd93a] "**Structural Modelling: An OO Framework and Development Process for Flight Simulators**", G. Abowd, L. Bass, L. Howard and L. Northrop, technical report number CMU-SEI-93-14, Software Engineering Institute, Carnegie Mellon University, 1993.

[Alexandridis86a] "**Adaptable Software and Hardware : Problems and Solutions**", Nikitas A. Alexandridis, in IEEE Computer, February 1986, p29-39.

[Arango91b] "**Domain Analysis Concepts and Research Directions**", Guillermo Arango and Rubén Prieto-Díaz, in "Domain Analysis and Software Systems Modelling", Prieto-Díaz, Rubén and Guillermo Arango (editors), IEEE Computer Society Press, p9-32, 1991.

[Arthur88a] "**Software evolution : the software maintenance challenge**", Lowell J. Arthur, New York : Wiley, 1988.

[Banerjee87a] "**Semantics and Implementation of Schema Evolution in Object-Oriented Databases**", Jay Banerjee, Won Kim, Hyong-Joo Kim and Henry F. Korth, in Proceedings of SIGMOD '87, 1987.

[Bass98a] "**Software Architecture in Practice**", Len Bass, Paul Clements and Rick Kazman, Addison-Wesley, ISBN 0-201-19930-0, 1998.

[Batory93a] "**Scalable Software Libraries**", Don Batory, Vivek Singhal, Marty Sirkin and Jeff Thomas, in The Proceedings of ACM SIGSOFT '93 : Symposium on the Foundations of Software Engineering, Los Angeles, California, 7-10 December, 1993.

[Bennett91a] "**Software Maintenance**", Keith H. Bennett, Barry J. Cornelius, Malcolm Munro and David J. Robson, in "Software Engineers' Reference Book", John McDermid (editor), ButterWorth-Heinemann, chapter 20, 1991.

[Bennett98a] "**Maintaining Maintainability**", Keith H. Bennett and Magnus Ramage, technical report, Research Institute in Software Evolution, Department of Computer Science, University of Durham, South Road, Durham, DH1 3LE, 1998.

[Bernstein95a] "**Living with Function Points**", Lawrence Bernstein and Alex Lubashevsky, technical report, AT&T, 1995.

[Blando98a] "**Modelling Behaviour with Personalities**", Luis Blando, Karl Lieberherr and Mira Mezini, technical report, College of Computer Science, Northeastern University, 360 Huntington Avenue, Boston, MA 02115 USA, 1998.

[Boehm76a] "**Software Engineering**", Barry W. Boehm, in IEEE Transactions on Computing, volume 25, p1226-1242, 1976.

[Booch91a] "**Object Oriented Design with Applications**", Grady Booch, The Benjamin/Cummins Publishing Company, Inc., 390 Bridge Parkway, Redwood City, California 94065, ISBN 0-8053-0091-0, 1991.

[Bosch97a] "**Superimposition: A Component Adaptation Technique**", Jan Bosch, technical report, University of Karlskrona/Ronneby, Department of Computer Science and

Business Administration, S-372 25 Ronneby, Sweden, 1997.

[Bowen86a] "**Meta-Level Techniques in Logic Programming**", K. Bowen, in Proceedings of the International Conference on Artificial Intelligence and its Applications, Singapore, 1986.

[Bradshaw96a] "**KaoS: An Open Agent Architecture Supporting Reuse, Interoperability, and Extensibility**", Jeffrey M. Bradshaw, technical report, Research and Technology, Boeing Information and Support Services, Seattle, WA 98024, 1996.

[Brownston95a] "**BBK Manual**", Lee Brownston, Knowledge Systems Laboratory, Dept. of Computer Science, Stanford University, Stanford, California 94305, September 1995.

[Bull95a] "**An Introduction to the WSL Program Transformer**", Tim Bull, technical report, Computer Science Department, University of Durham, South Road, Durham, DH1 3LE, UK, January 1995.

[Cazzola97a] "**Architectural Reflection : Bridging the Gap Between a Running System and its Architectural Specification**", W. Cazzola, A. Savigni, A. Sosio and F. Tisato, technical report, DSI – University of Milan, Via Comelico 39-41, 20135 Milan, Italy, September 1997.

[Chastek96a] "**A Case Study in Structural Modelling**", G. Chastek and L. Brownsword, technical report number CMU/SEI-96-TR-35, Software Engineering Institute, Carnegie-Mellon University, December 1996.

[Chen76a] "**The Entity-Relationship Model – Toward a Unified View of Data**", P. P. Chen, in ACM Transactions on Database Systems, 1 (1), p9-36, March 1976.

[Cimio95a] "**ISO TC184/SC4/WG5 N243, EXPRESS-M Reference Manual**", CIMIO Ltd., July 1995.

[Clamen94a] "**Schema Evolution and Database Conversion Support Exhibited by Known Research and Commerical DB and OODBMS Version 3.4**", Stewart M. Clamen, http://www.cs.cmu.edu/People/clamen/work/evolution-summary.gz, 1994.

[Codd70a] "**A Relational Model of Data for Large Shared Data Banks**", E.F. Codd, in Communications of the ACM, volume 13, p377-387, 1970.

[Codd82a] "**Relational Database: a Practical Foundation for Productivity**", E.F. Codd, in Communications of the ACM, volume 25, p109-117, 1982.

[ComponentGlossary] "**Component Software Glossary**".

[Corkill91a] "**Blackboard Systems**", Daniel D. Corkill, Blackboard Technology Group, Inc., in AI Expert, 6 (9), p40-47, September 1991.

[Crelier95a] "**Extending Module Interfaces without Invalidating Clients**", Régis Crelier, in Software - Concepts and Tools, volume 16, p49-62, 1995.

[DeBaud94a] "**Domain Analysis and Reverse Engineering**", Jean-Marc DeBaud, Bijith Moopen and Spencer Rugaber, in Proceedings of ICSM '94, p326-335, 1994.

[Denno96a] "**Dynamic Objects and Meta-Level Programming of an EXPRESS Language Environment**", Peter Denno, technical report, Manufacturing Systems Integration Division, National Institute of Standards and Technology, Gaithersburg, Maryland,

USA, 1996.

[Finin93a] "**Draft Specification of the KQML Agent Communication Language**", Tim Finin, Jay Weber, Gio Wiederhold, Michael Genesereth, Richard Fritzon, Donald McKay, James McGuire, Richard Pelavin, Stuart Shapiro and Chris Beck, 15th June 1993.

[Finkelstein90a] "**ViewPoint Oriented Software Development**", Anthony Finkelstein, Jeff Kramer and Michael Goedicke, in Proceedings of WSEA '90, 1990.

[Finkelstein92a] "**Viewpoints: A Framework for Integrating Multiple Perspectives in System Development**", A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein and M. Goedicke, in International Journal of Software Engineering and Knowledge Engineering, 2 (1), p31-58, March 1992.

[Finkelstein94a] "**Inconsistency Handling in Multi-Perspective Specifications**", A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer and B. Nuseibeh, in IEEE Transactions on Software Engineering, 20 (8), p569-578, August 1994.

[Flanagan97a] "**Java in a Nutshell**", David Flanagan, O'Reilly and Associates, 1997.

[Forrest91a] "**Emergent Computation**", Stephanie Forrest, MIT Press, 1991.

[Franz95a] "**Protocol Extension: A Technique for Structuring Large Extensible Software Systems**", Michael Franz, in Software - Concepts and Tools, volume 16, p86-94, 1995.

[Freed96a] "**MIME Part 1: Format of Internet Message Bodies**", N. Freed and N. Borenstein, RFC 2045, November 1996.

[Freed96b] "**MIME Part 2: Media Types**", N. Freed and N. Borenstein, RFC 2046, November 1996.

[Freed96c] "**MIME Part 4: Registration Procedures**", N. Freed, J. Klensin and J. Postel, RFC 2048, November 1996.

[Freed96d] "**MIME Part 5: Conformance Criteria and Examples**", N. Freed and N. Borenstein, RFC 2049, November 1996.

[Garlan93a] "**An Introduction to Software Architecture**", David Garlan and Mary Shaw, in "Advances in Software Engineering and Knowledge Engineering", V. Ambriola and G. Tortora (editors), World Scientific Publishing Company, New Jersey, January 1993.

[Gelernter92a] "**Coordination Languages and their Significance**", David Gelernter and Nicholas Carriero, in Communications of the ACM, 35 (2), p97-107, February 1992.

[Genesereth94a] "**Software Agents**", Michael R. Genesereth and Steven P. Ketchpel, in Communications of the ACM, 37 (7), July, 1994.

[Ginsberg94a] "**Artificial Intelligence and Non-Monotonic Reasoning**", Matthew L. Ginsberg, in "Handbook of Logic in Artificial Intelligence and Logic Programming Volume 3 Nonmonotonic Reasoning and Uncertain Reasoning", Oxford Science Publications, Dov M. Gabbay, C. J. Hogger and J. A. Robinson (editors), p1-28, 1994.

[Goldman95a] "**The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications**", Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson and Ram Sethuraman, in IEEE Transactions on Software Engineering, 21 (9), p735-746, September 1995.

[Gruber93a] "**Toward Principles for the Design of Ontologies Used for Knowledge Sharing**", Thomas R. Gruber, in "Formal Ontology in Conceptual Analysis and Knowledge Representation", Nicola Guarino and Roberto Poli (editors), Kluwer Academic Publishers, 1993.

[Halstead77a] "**Elements of Software Science**", M.H. Halstead, Amsterdam: North-Holland, 1977.

[Hansen90a] "**The C++ Answer Book**", Tony L. Hansen, Addison-Wesley Publishing Company, ISBN 0-201-11497-6, 1990.

[Hillis98a] "**The Pattern on the Stone**", W. Daniel Hillis, Weidenfeld and Nicolson, 1998.

[Holland98a] "**Emergence: From Chaos to Order**", John H. Holland, Addison-Wesley, ISBN 0-201-14943-5, 1998.

[Hursch95a] "**Maintaining Consistency and Behaviour of Object-Oriented Systems During Evolution**", Walter L. Hursch, Ph.D. thesis, College of Computer Science, Northeastern University, Boston, Massachusetts, USA, August 1995.

[Hursch95b] "**Separation of Concerns**", Walter L. Hursch and Cristina Videira Lopes, technical report, College of Computer Science, Northeastern University, Boston, MA 02115, USA, February 1995.

[Jacobson92a] "**Object-Oriented Software Engineering: A Use Case Driven Approach**", Ivar Jacobson, Magnus Christerson, Patrik Jonsson and Gunnar Övergaard, Addison-Wesley, ISBN 0-201-54435-0, 1992.

[Jantsch80a] "**The Self-Organising Universe: Scientific and Human Implications of the Emerging Paradigm of Evolution**", E. Jantsch, 1980.

[Jones86a] "**Systematic Software Development Using VDM**", C. B. Jones, Prentice-Hall, 1986.

[Kanada94a] "**Stochastic Problem Solving by Local Computation based on Self-Organization Paradigm**", Yasusi Kanada and Masao Hirokawa, in Proceedings of ICSS '94, p82-91, 1994.

[Karakostas90a] "**Modelling and Maintenance Software Systems at the Teleological Level**", V. Karakostas, in Software Maintenance: Research and Practice, volume 2, p47-59, 1990.

[Keller96a] "**Change Costing in a Maintenance Environment**", Ted W. Keller, Keynote Speech in Proceedings of ICSM '96, November 1996.

[Kiczales91a] "**The Art of the Metaobject Protocol**", Gregor Kiczales, Jim des Rivieres and Daniel G. Bobrow, ISBN 0-262-11158-6 hardback 0-262-61074-4 paperback, MIT Press, 1991.

[Kiczales92a] "**Towards a New Model of Abstraction in the Engineering of Software**", Gregor Kiczales, in Proceedings of IMSA '92, 1992.

[Kiczales93b] "**Metaobject Protocols: Why we want them and what else they can do**", Gregor Kiczales, J. Michael Ashley, Luis Rodriguez, Amin Vahdat and Daniel G. Bobrow, in "Object-Oriented Programming: The CLOS Perspective", A. Paepcke (editor), The MIT Press, Cambridge, MA, p101-118, 1993.

[Kiczales96a] "**Beyond the Black Box: Open Implementation**", Gregor Kiczales, in IEEE Software, 13 (1), January 1996.

[Kiczales96b] "**Open Implementation and Metaobject Protocols**", Gregor Kiczales and Andreas

Paepcke, technical report, Xerox Palo Alto Research Centre, 3333 Coyote Hill Road, Palo Alto, CA, 94304, USA, 1996.

[Kiczales97a] "**Aspect-Oriented Programming**", Gregor Kiczales, J. Lamping et al, in Proceedings of OOPSLA '97, 1997.

[Kiczales97b] "**Open Implementation Design Guidelines**", Gregor Kiczales, John Lamping, Cristina Videira Lopes, Chris Madea, Anurag Mendhekar and Gail Murphy, technical report, Xerox Palo Alto Research Centre, 3333 Coyote Hill Road, Palo Alto, CA, 94304, USA, 1997.

[Kishimoto95a] "**Adapting Object-Communication Methods Dynamically**", Yoshinori Kishimoto, Nobuto Kotaka and Shinichi Honiden, in IEEE Software, 11 (3), p65, May 1995.

[Kotula] Jeff Kotula, Patterns Discussion Group Email.

[Laird86a] "**Chunking in SOAR: The Anatomy of a General Learning Mechanism**", J. Laird, P. Rosenbloom and A. Newell, in Machine Intelligence, 1 (1), Kluwer Academic Publishers, 1986.

[Lalanda97a] "**A Control Model for the Dynamic Selection and Configuration of Software Components**", Philippe Lalanda, technical report, Thomson-CSF Corporate Research Laboratory, Domaine de Corbeville, F-91404 Orsay, France, 1997.

[Lehman85a] "**Programs, Life Cycles and Laws of Software Evolution**", M.M. Lehman, in "Program Evolution : Processes of Software Change", M. M. Lehman and L.A. Belady (editors), Academic Press Inc. Ltd., 24-28 Oval Road, London, NW1 7DX, p393-450, 1985.

[Lehman85b] "**Program Evolution : Processes of Software Change**", M. M. Lehman, and L. A. Belady, Academic Press Inc. Ltd., 24-28 Oval Road, London, NW1 7DX, ISBN 0-12-442440-6, 1985.

[Lehman98a] "**On Evidence Supporting the FEAST Hypothesis and the Laws of Software Evolution**", M. M. Lehman, D. E. Perry and J. F. Ramil, in Proceedings Metrics 1998, Bethesda, Maryland, November 20-21, 1998.

[Lehman99a] "**FEAST, FEAST/1 and FEAST/2**", M. M. Lehman, departmental talk, Research Institute in Software Evolution, Department of Computer Science, University of Durham, DH1 3LE, January 1999.

[Lieberherr93a] "**Object-Oriented Software Evolution**", Karl Lieberherr and Cun Xiao, in IEEE Transactions on Software Engineering, 19 (4), p313-343, April 1993.

[Lieberherr94a] "**Adaptive Object-Oriented Programming Using Graph-Based Customization**", Karl J. Lieberherr, Ignacio Silva-Lepe and Cun Xiao, Communications of the ACM, 37 (5), p94-101, May 1994.

[Lieberherr95a] "**Workshop on Adaptable and Adaptive Software**", Karl J. Lieberherr, in Addendum to the Proceedings of OOPSLA 1995, Austin, Texas, S. Bilow and P. Bilow (editors), ACM Press, p149-154, 1995.

[Lieberherr96a] "**Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns**", Lieberherr, Karl J., PWS Publishing Company, Boston, January 1996, ISBN

0-534-94602-X.

[Liebich95a] "**A Survey of Mapping Methods Available Within the Product Modelling Arena**", Thomas Liebich, Robert Amor and Marcel Verhoef, technical report, Delft University of Technology, Faculty of Civil Engineering, Building Engineering Group, P.O. Box 5048, 2600 GA Delft, The Netherlands, 1995.

[Lopes94a] "**Thinking Adaptively : the Demeter System**", Cristina Videira Lopes and Karl Lieberherr, technical report, College of Computer Science, Northeastern University, June 1994.

[Lopes97a] "**D: A Language Framework for Distributed Programming**", Cristina Videira Lopes and Gregor Kiczales, technical report, Xerox Palo Alto Research Centre, 333 Coyote Hill Road, Palo Alto, CA 94304, USA, 1997.

[Maes87a] "**Concepts and Experiments in Computational Reflection**", Pattie Maes, in Proceedings of OOPSLA '87, 1987.

[Maes87b] "**Computational Reflection**", Pattie Maes, Ph.D. thesis, Vrije University, Brussels, Belgium, 1987.

[McCabe76a] "**A Complexity Measure**", T.J. McCabe, in IEEE Transactions on Software Engineering, 2 (4), p308-320, 1976.

[McDermid91a] "**Software Engineers' Reference Book**", John McDermid (editor), ButterWorth-Heinemann, 1991.

[Meek95a] "**What is a Procedure Call?**", Brian L. Meek, in ACM SIGPLAN Notices, 30 (9), p33-40, September 1995.

[Mendhekar97a] "**RG: A Case Study for Aspect-Oriented Programming**", Anurag Mendhekar, Gregor Kiczales and John Lamping, technical report, Xerox Palo Alto Research Centre, 3333 Coyote Hill Road, Palo Alto, CA 94304, USA, 1997.

[Mikhajlov97a] "**The Fragile Base Class Problem and its Solution**", Leonid Mikhajlov and Emil Sekerinski, technical report, Turku Centre for Computer Science, University of Turku, Lemminkäisenkatu 14, FIN-20520 Turku, Finland, 1997.

[Osborn93a] "**Information Systems Lessons Learned**", Lloyd Osborn, in Educating the Next Generation of Information Specialists, Alexandria, VA, 1993, National Science Foundation, p40-41, 1993.

[Ossher94a] "**Subject-Oriented Programming: Supporting Decentralised Development of Objects**", Harold Ossher, William Harrison, Frank Budinsky and Ian Simmonds, in the Proceedings of the 7th IBM Conference on Object-Oriented Technology, July, 1994.

[Parnas72a] "**On the Criteria to Be Used in Decomposing Systems into Modules**", David L. Parnas, in Communications of the ACM, 15 (2), p1053-1058, 1972.

[Peterson94a] "**Mapping a Domain Model and Architecture to a Generic Design**", A. Spencer Peterson and Jay L. Stanley Jr., technical report, Carnegie Mellon University Software Engineering Institute, May 1994.

[Potter91a] "**An Introduction to Formal Specification and Z**", Ben Potter, Jane Sinclair and David Till, Prentice-Hall, 1991.

[Pountain94a] "**Extensible Software Systems**", Dick Pountain and Clemens Szyperski, in Byte, May 1994.

[Rumbaugh91a] "**Object-Oriented Modelling and Design**", J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy and W. Lorensen, Prentice-Hall, NJ, 1991.

[Schenck94a] "**Information Modelling: The EXPRESS Way**", Douglas Schenck and Peter Wilson, Oxford University Press, 1994.

[Schieffer93a] "**Supporting Integration and Evolution with Object-Oriented Views**", Bernhard Schieffer, technical report, Forschungszentrum Informatik (FZI), Haid-und-Neu-Strasse 10-14, D-76131 Karlsruhe, 1993.

[Seiter96a] "**Evolution of Object Behaviour Using Context Relations**", Linda M. Seiter, Jens Palsberg and Karl J. Lieberherr, in "Symposium on Foundations of Software Engineering", David Garlan (editor), ACM Press, 1996.

[Seiter98a] "**Evolution of Object Behaviour Using Context Relations**", Linda M. Seiter, Jeng Palsberg and Karl Lieberherr, in IEEE Transactions on Software Engineering, 24 (1), January 1998, p79-92.

[Shaw96a] "**Software Architecture: Perspectives on an Emerging Discipline**", Mary Shaw and David Garlan, Prentice-Hall Inc., Upper Saddle River, New Jersey 07458, ISBN 0-13-182957-2, 1996.

[Simonyi96a] "**Intentional Programming – Innovation in the Legacy Age**", Charles Simonyi, in Proceedings of IFIP Working Group 2.1 Meeting, June 1996.technical report, Microsoft Corporation.

[Skarmeas97a] "**Intelligent Routing Based on Active Patterns as the Basis for the Integration of Distributed Information Systems**", Nikolaos Skarmeas and Keith L. Clark, technical report, Department of Computing, Imperial College, London, UK, February 1997.

[Smith82a] "**Reflection and Semantics in a Procedural Language**", Brian Smith, technical report, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1982.

[Smith93a] "**Formally Describing Objects and Systems : Getting Some Use Out of the Distinction Between Internal and External Descriptions**", Simon Smith, in Proceedings of Object Oriented Programming Systems Languages and Architectures (OOPSLA), p141, 1993.

[Smith95a] "**Is Maintenance Ready for Evolution?**", Simon Smith, Keith Bennett and Cornelia Boldyreff, in the Proceedings of the International Conference on Software Maintenance, Opio (Nice), France, October 17-20, 1995, p367-372.

[Smith96a] **Departmental Talk** by Simon Smith, Department of Computer Science, University of Durham, South Road, Durham, DH1 3LE, 1996.

[Steele90a] "**Common Lisp: The Language, Second Edition**", Guy Steele, Digital Press, 1990.

[Stroud92a] "**Transparency and Reflection in Distributed Systems**", Robert Stroud, technical report, Department of Computing Science, University of Newcastle Upon Tyne, Newcastle, UK, April 1992.

[Stroustrup94a] "**The Design and Evolution of C++**", Bjarne Stroustrup, Addison-Wesley Publishing Company, ISBN 0-201-54330-3, 1994.

[Tanenbaum96a] "**Computer Networks Third Edition**", Andrew S. Tanenbaum, Prentice-Hall International, Inc., 1996.

[Templ93a] "**A Systematic Approach to Multiple Inheritance Implementation**", J. Templ, in ACM SIGPLAN Notices, 28 (4), p61-66, April 1993.

[Tepfenhart97a] "**A Unified Object Topology**", William M. Tepfenhart and James J. Cusick, in IEEE Software, p31-35, January 1997.

[Thompson92a] "**The Pocket Oxford Dictionary**", Della Thompson (editor), Oxford University Press, 1992.

[Tresch93a] "**Schema Transformation Without Database Reorganization**", Markus Tresch and Marc H. Stoll, in ACM SIGMOD Record, 22 (1), March 1993.

[Turver93a] "**Early detection of Ripple Propagation in Evolving Software Systems**", Richard J. Turver, Ph.D. thesis, Department of Computer Science, University of Durham, South Road, Durham, DH1 3LE, 1993.

[Ward94a] "**Language-Oriented Programming**", Martin Ward, in Software-Concepts and Tools volume 15, p147-161, 1994.

[Wolverton94a] "**BB1 v3.2 Manual**", Michael Wolverton and Lee Brownston, technical report number KSL 94-XX, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, Stanford, California 94305, March 1994.

[Yau78a] "**Ripple Effect Analysis of Software Maintenance**", S. S. Yau, J.S. Collofello and T. MacGregor, in Proceedings of the IEEE COMPSAC, November 1978, p60-65.

[Zibman95a] "**An Architectural Approach to Minimising Feature Interaction in Telecommunications**", Israel Zibman, Carl Woolf, Peter O'Reilly, Larry Strickland, David Willis and John Visser, technical report, GTE Laboratories, Inc., January 1995.